MASTER EN SCIENCES INFORMATIQUES

Enrichissement d'un diagramme de classes par design patterns selon une approche transformationnelle

Barbieux, Steve

Award date:
2006

Faculté Universitaire Notre-Dame de la Paix, Namur
Institut d'Informatique

Année académique 2005 – 2006

Enrichissement d'un diagramme de classes par design patterns selon une approche transformationnelle

Steve Barbieux

Mémoire présenté en vue de l'obtention du grade de Licencié en Informatique

Résumé

Les design patterns constituent des solutions éprouvées et documentées à des problèmes récurrents dans le domaine de la programmation orientée objet.

Des outils CASE proposent ces design patterns sous forme de librairies de diagrammes de classes qui peuvent être réutilisés dans d'autres diagrammes de classes. Néanmoins, l'apport de ces outils reste limité et donc insuffisant.

En effet, nous pensons que l'enrichissement d'un diagramme à l'aide d'un design pattern devrait être spécifique à ce design pattern et transformer le diagramme en conséquence. Nous proposons donc un prototype de système de transformation de diagrammes de classes à l'aide de design patterns que nous critiquons avant de suggérer des pistes d'amélioration et des alternatives.

Le coeur de cette implémentation est réalisé en Prolog et l'interface utilisateur est assurée par DB-Main programmé par son langage intégré Voyager 2.

Mots clefs

Modèles de conception (design patterns), Prolog, diagramme de classes, outil CASE, transformation.

Abstract

The design patterns constitute tested and documented solutions to recurring problems in object oriented programming area.

CASE tools propose these design patterns in the form of library of class diagrams which can be re-used in other class diagrams. Nevertheless, the contribution of these tools remains limited and therefore insufficient.

Indeed, we think that the enrichment of one diagram using a design pattern should be specific to this design pattern and should transform the diagram thereof. We thus propose a prototype of system of transformation of class diagrams using design patterns which we criticize before suggest tracks of improvement and alternatives.

The heart of this implementation is carried out in Prolog and the user interface is provided by DB-Main programmed by its integrated language Voyager 2.

Keywords

Design Patterns, Prolog, Class Diagram, CASE tool, transformation.

Nous tenons à remercier notre promoteur, Vincent Englebert, pour son apport dans la réalisation de ce mémoire. Non seulement, il a proposé l'idée originale défendue dans ce texte, mais il a été capable de nous communiquer son enthousiasme pour le sujet.

Par ailleurs, nous remercions l'ensemble du corps professoral et facultaire de la licence en informatique en horaire décalé. Puisse cette initiative indispensable vivre encore de longues années.

Enfin, plusieurs personnes nous ont aidé de diverses manières dans la réalisation de ce texte et nous souhaitons toutes les en remercier. En particulier, nos pensées vont vers Sophie De Muynck qui a fait preuve de plus d'effort et de patience que nous n'aurions pu le souhaiter et sans qui ce mémoire eut été tout autre. Pour cela et bien d'autres choses, merci.

Table des matières

Table des figures

Glossaire

CASE tool ou outil CASE : "CASE tool" ou "outil CASE" comprend l'acronyme de Computer-Aided Software Engineering. "Un CASE tool est un produit informatique visant au support d'une ou plusieurs activités de génie logiciel dans un processus de développement logiciel."[1]

Composite : COMPOSITE est un design pattern de type structurel. "Le modèle Composite compose des objets en des structures arborescentes pour représenter des hiérarchies composant/composé. Il permet au client de traiter de la même et unique façon des objets individuels et les combinaisons de ceux-ci."[GoFDP, p. 189]

Décorateur : DÉCORATEUR est un design pattern de type structurel. Il est utilisé afin d'ajouter dynamiquement des responsabilités supplémentaires à un objet de manière alternative à l'héritage.

Design Patterns ou DP : les design patterns sont des modèles de conceptions réutilisables pour la programmation orientée-objet. Les vingt-trois plus célèbres sont ceux introduits par le GoF dans le livre *Design Patterns [GoFDP].*

Fabrique abstraite : FABRIQUE ABSTRAITE est un design pattern de type créateur. "Fabrique abstraite fournit une interface pour la création de familles d'objets apparentés ou interdépendants, sans qu'il soit nécessaire de spécifier leurs classes concrètes."[GoFDP, p. 101]

MDA : Model Driven Architecture (MDA `http://www.omg.org/mda/`) est développé par l'OMG. "MDA est une approche pour utiliser des modèles dans le dévelopement logiciel." [MDAG, p. 2-1]

Méta : méta est un préfixe issu du grec $\mu\varepsilon\tau\alpha$ qui signifie (entre autres) : "Mét(a)- exprime une idée de transcendance ; les mots constr., gén. subst., appartiennent le plus souvent aux domaines de la philos. et des sc. hum. ; ils signifient «(ce) qui est au-delà de, ce qui dépasse et englobe la réalité désignée par le 2e élém.»"[TLFi, "Mét(a)-, élém. formant"][2]

Modèle entité-association ou ERA : le modèle entité-association est aussi appelé Entité-Relation-Attribut, d'où l'acronyme ERA. Ce modèle permet de décrire un domaine de données – habituellement en vue de l'intégration des ces données dans une base de données.

---

[1] Carnagie Melon Software Engineering Institute*[CMSEI]*, "Computer Aided Software Engineering (CASE) Environment" *in* `http://www.sei.cmu.edu/legacy/case/case_whatis.html` (consulté le 4 mai 2006).

[2] *TLFi*,"Mét(a)-,élém. formant" *in* Méta `http://atilf.atilf.fr/dendien/scripts/tlfiv5/visusel.exe?13;s=3158554440;r=1;nat=;sol=2;` (Consulté le 21 avril 2006)

OMG : "Object Management Groupe est un consortium à but non lucratif qui produit et maintient des spécifications pour l'industrie informatique." [OMG] L'OMG a notamment développé MDA.

Prolog : Prolog est un langage de programmation qui s'inscrit dans le paradigme de programmation logique. Le nom de ce langage vient de PROgramming in LOGic.

Singleton : SINGLETON est un design pattern de type créateur ; il est sans doute le plus simple et le plus utilisé des design patterns. SINGLETON permet d'accéder globalement à l'instance, garantie unique, d'une classe.

Stratégie : STRATÉGIE est un design pattern de type comportemental. Il "définit une famille d'algorithmes, encapsule chacun d'entre eux, et les rend interchangeables. Le modèle Stratégie permet aux algorithmes d'évoluer indépendamment des clients qui les utilisent."[GoFDP, p. 369]

UML : UML est l'acronyme de *Unified Modeling Language*. "Le langage de modélisation unifié (UML) est un langage pour spécifier, visualiser, construire et documenter les artefacts des système logiciels aussi bien que les modélisation commerciale (*business*) et autre systèmes non-logiciels."[UML, p. 1-1] Ce langage définit notamment le diagramme de classes comme modèle de la structure statique d'un système, en particulier d'un programme orienté objet.

Voyager 2 : "Voyager 2 est un langage impératif avec des caractéristiques originales telles le type primitif liste avec garbage collection et des requêtes déclaratives sur le référentiel (*repository*) prédéfini de l'outil DB-Main."[V2, p. 3]

Première partie

Partie préliminaire

Chapitre 1

Introduction générale

La conception d'un logiciel orienté objet suppose de nombreux choix aux nombreuses conséquences. Dans cette tâche délicate, des modèles de conception réutilisables fournissent une aide précieuse : les design patterns font bénéficier leur utilisateur de l'expérience de ceux qui, avant lui, ont réfléchi à la meilleure exploitation possible des avantages du concept orienté objet.

Un élément central de cette conception est le diagramme de classes, modèle représentant la structure statique du système à concevoir. Par ailleurs, ce diagramme peut être enrichi au moyen de design patterns ; dans cet objectif, le concepteur peut avoir recours à des outils d'aide à la conception : ces derniers disposent d'un catalogue de diagrammes de classes correspondant aux design patterns et peuvent en enrichir un diagramme particulier.

Les processus d'enrichissement en question restent néanmoins limités : nous en proposons une alternative qui utilise Prolog comme langage de programmation pour ce que que nous appellerons une approche transformationnelle des diagrammes de classes. Cette réalisation requiert une interface utilisateur : c'est DB-Main qui, comme outil graphique et interactif, joue ce rôle.

Les chapitres qui suivent discutent cette alternative : ainsi, dans notre étude de la question, nous fixons les bases des notions utilisées et explicitons notre propos en nous attachant en particulier au concept de design pattern.

Ensuite, sous l'angle qui nous occupe, nous analysons trois logiciels d'aide à la conception : DPA Toolkit, Enterprise Architect et Borland Together. Leur critique nous permet de cerner les avantages et inconvénients auxquels nous pourrons comparer nos résultats.

Notre troisième partie, la plus développée, expose notre proposition. Dans le chapitre 4, "Approche transformationnelle", nous justifions la pertinence du choix de Prolog comme langage de programmation en exposant les qualités de la programmation déclarative, celle de l'unification et enfin de la méta-programmation que permet ce langage. Puis, nous décrivons notre solution dans le chapitre 5 en explicitant notre modèle de système de connaissances, notre conception physique et l'intégration dans l'interface visuelle que constitue DB-Main. Nous complétons cette description d'une présentation détaillée d'une

transformation à l'aide d'un design pattern choisi et terminons cette partie par la liste des transformations effectivement implémentées.

Pour rendre ces notions théoriques plus parlantes, nous décrivons un exemple d'utilisation concrète du système proposé : ainsi, une étude de cas montre l'enrichissement à l'aide de cinq design patterns d'un diagramme de classes par l'application successive des cinq transformations que nous avons développées.

Fort des notions et illustrations ainsi exposées, nous pouvons critiquer notre solution, notamment à la lumière de l'étude de logiciels existant ; nous proposons alors quelques pistes d'amélioration de notre système de transformation et des alternatives à nos choix technologiques.

Nous invitons également le lecteur à compléter ces chapitres, d'une part, par les annexes à notre étude, parties significatives de la programmation de notre solution ; d'autre part, par un glossaire situé en début de texte épinglant les définitions-clefs de nos recherches. Enfin, nous signalons que, par souci de clarté, nous avons choisi de ne noter que des références bibliographiques abrégées, et renvoyons pour les précisions concernant les références essentielles à la bibliographie finale.

Chapitre 2

La question

Comprendre la question que nous abordons présuppose quelques connaissances spécifiques. Nous tenons pour acquises les notions de programmation orientée-objet telles l'héritage, l'abstraction, la visibilité ou le polymorphisme ainsi que le concept et la syntaxe des diagrammes de classes[1].

Néanmoins, nous discutons, dans un premier temps, de la conceptualisation de cette structure statique d'un système et des qualités que nous souhaitons que cette dernière insuffle dans cette même structure.

Dans un second temps, nous introduisons la notion de design pattern appuyée d'un exemple ; nous présentons ensuite le concept d'enrichissement d'un diagramme de classes à l'aide de design patterns.

Fort du contexte ainsi balisé, nous clôturons ce premier propos en précisant l'objet de nos recherches.

2.1 Modélisation à l'aide d'un diagramme de classes

Avant de commencer un programme dans un langage orienté-objet, une bonne pratique voudrait que le concepteur commence par dessiner un diagramme de classes. Ce diagramme est le plan statique du programme qui servira de base à la programmation. Il représente les objets fondamentaux du système, leurs caractéristiques et leurs relations.

Toutefois, un diagramme de classes peut représenter bien plus qu'un plan de programme en ce qu'il est aussi un type de modèle qui pourra être utilisé de manière plus ou moins abstraite pour représenter un modèle de données. Il sert aussi couramment à modéliser un programme, mais aussi une base de données ou pourquoi pas le modèle conceptuel d'un domaine étudié.

Dans le cadre de notre recherche, nous entendons le diagramme de classes comme la structure statique d'un système informatique destiné à être programmé dans un language orienté-objet.

[1] Ces notions ont été présentées dans le cours du professeur B. Le Charlier, *Principes des Langages de Programmation : Paradigmes de Programmation (I)* [PdP1].

Le diagramme de classes est la structure statique la plus connue de l'UML (Unified Modeling Language) dont on peut trouver la définition formelle dans les documents[2] publiés par l'OMG (Object Management Group). Une initiation plus simple passera par un des nombreux ouvrages sur le sujet[3].

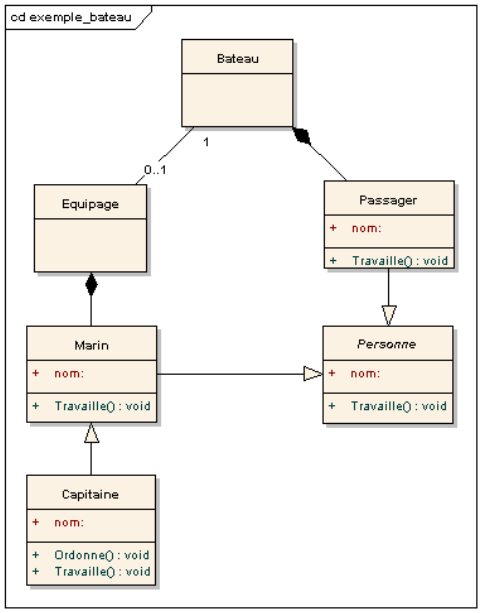

FIG. 2.1: Exemple de diagramme de classes

Précisons que si notre recherche ne s'attachera pas à définir le diagramme de classes, elle proposera au lecteur un méta-modèle simplifié d'un diagramme de classes (voir 5.1) qui permettra de comprendre les éléments abstraits que nous manipulons [4] et leurs interactions.

Enfin, concevoir le diagramme de classes adéquat d'un système est difficile à plus d'un titre : le concepteur cherchera bien sûr à lui donner la lisibilité et la clarté d'un bon modèle ; toutefois, sachant que de ce modèle dépendront en grande partie les qualités du résultat, c'est-à-dire celles du système implémenté, le concepteur cherchera à réaliser une structure qui non seulement puisse produire un résultat fonctionnel, mais induise également au programme des qualités annexes importantes.

Les propriétés non-fonctionnelles désirées sont nombreuses. Citons par exemple :

- L'adaptabilité :
  - les spécifications peuvent évoluer tout au long de la vie du système ; un système adaptable pourra plus facilement se voir ajouter des fonctionnalités ou voir transformer les fonctionnalités existantes. Bien plus, pour modifier

[2] Pour l'UML version 1.5 (une version 2.0 existe à présent) : Object Management Group, OMG. UML version 1.5, document number : formal/03-03-01, mars 2003.

[3] Citons par exemple *Instant UML* de Pierre-Alain Muller [IUML] ou pour une référence bien plus concise, l'annexe B "Guide de notation" du livre *Design Patterns* [GoFDP].

[4] Classe, association, généralisation, attribut, opération, etc.

un système avec une mauvaise adaptabilité, le programmeur devra parfois recommencer entièrement un pan de son travail.
- La portabilité :
  - porter un système d'une plateforme à une autre peut nécessiter beaucoup d'énergie. Pour porter le système d'une plateforme particulière vers une autre plateforme particulière, il est intéressant de prévoir une architecture qui sépare la partie spécifique à la première plateforme particulière du reste, plus générique, de l'application, afin de ne devoir réaliser à nouveau que cette partie particulière.
- La réutilisabilité :
  - but ultime de la programmation orientée objet, la réutilisabilité est la possibilité de réutiliser des morceaux de logiciels. En effet, programmer en réutilisant des composants bien testés et parfois difficiles à mettre au point permet de gagner beaucoup de temps. Un programmeur souhaitera donc d'une part utiliser le travail fait par d'autres (tels des composants standards), mais aussi profiter à l'avenir de ce qu'il aura lui-même mis au point et réaliser des composants réutilisables.

Les pages qui suivent vont montrer qu'une bonne conception rencontre un atout majeur dans l'utilisation des design patterns. Ceux-ci faciliteront la mise au point de systèmes présentant les qualités désirées.

2.2 Design patterns[5]

2.2.1 Présentation du concept

Une bonne conception d'un logiciel orienté-objet nécessite de l'expérience surtout si on cherche à réellement profiter de la puissance de l'orienté-objet ; un débutant tombe souvent, nous semble-t-il, dans le piège d'utiliser un langage orienté-objet comme un language impératif dont il a généralement une plus grande habitude.

Les design patterns résolvent ce clivage : ils présentent des modèles de conception "bien connus" nous faisant profiter de l'expérience de nos prédécesseurs. Ainsi, tout concepteur de logiciel orienté-objet même débutant dispose de solutions expérimentées et améliorées pendant des années par les meilleurs de leurs collègues.

Redde Caesari quae sunt Caesaris : cette avancée dans le domaine du génie logiciel fut introduite par quatre informaticiens aujourd'hui mondialement célèbres sous le nom de *Gang of Four*[6] ou GoF[7]. Ils mirent au point et documentèrent

[5] **Les modèles de conception qui nous occupent seront invariablement appelés Design Patterns ou DP.**

[6] Le *Gang of Four* : Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides.

[7] Le concept de modèle de conception est antérieur à son utilisation dans l'informatique. Christopher Alexander en avait déjà donné une définition qui s'appliquait à l'architecture et l'urbanisme dans son livre *A Pattern Language* dont l'ambitieux objectif était de permettre à chacun de pouvoir concevoir et construire sa propre maison. (C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King and S. Angel, *A Pattern Language*, Oxford University Press, New York, 1977.)

vingt-trois modèles permettant de résoudre vingt-trois problèmes classiques de conception. Bien d'autres design patterns furent créés depuis, mais ces vingt-trois-là restent les plus connus et les plus utilisés (parfois inconsciemment) par les concepteurs de programmes orientés-objet.

Plus précisémment, un design pattern est composé de 4 éléments : le nom de modèle, le problème, la solution, les conséquences.

> "1. Le nom de modèle est un moyen de décrire en un ou deux mots un problème de conception, ses solutions, et leurs conséquences. Donner un nom à un modèle accroît immédiatement le vocabulaire de conception. Cela permet de travailler à un degré d'abstraction plus élevé. Disposer d'un vocabulaire pour les modèles permet de les citer aux collègues, dans une documentation ou même à soi-même. Cela facilite la recherche de solutions, ainsi que leur communication à des tiers, nanties de leur variantes de compromis. [...]
>
> 2. Le problème décrit les situations où le modèle s'applique. Il expose le sujet à traiter et son contexte. Il peut s'agir de problèmes spécifiques de conception comme, par exemple, la représentation d'algorithmes comme des objets. Il peut décrire des structures de classes ou d'objets, typiques d'une conception immuable. Le problème comportera parfois une liste de conditions à satisfaire pour que le modèle s'applique valablement.
>
> 3. La solution décrit les éléments qui constituent la conception, les relations entre eux, leur part dans la solution, leur coopération. La solution ne décrit pas un modèle précis, ni une implémentation, puisqu'un modèle est une sorte de patron qui s'applique dans diverses situations. Le modèle fournit plutôt la description générique d'un problème de conception, et indique comment un agencement d'éléments (dans notre cas, des classes et des objets) peut le résoudre.
>
> 4. Les conséquences sont les effets résultant de la mise en oeuvre du modèle et les variantes de compromis que celle-ci entraîne. Bien que ces conséquences soient rarement évoquées lors de la description des choix de conception, elles sont déterminantes pour l'évaluation des alternatives de conception et pour l'appréciation des avantages et des inconvénients de l'application du modèle.
> [...] Etant donné que la réutilisation est souvent le facteur déterminant d'une conception orientée-objet, les conséquences d'un modèle incluent son impact sur la flexibilité d'un système, son extensibilité, ou sa portabilité. Faire la liste explicite des conséquences permet de les comprendre et de les évaluer."
>
> [GoFDP, p. 3-4]

Un modèle de conception est un terme assez générique et pourrait couvrir bien des cas. Dans notre contexte, nous nous attacherons à des modèles de conception qui "font la description d'objets coopératifs et de classes que l'on a spécialisés pour résoudre un problème général de conception dans un contexte particulier." [GoFDP, p. 4]

2.2.2 Description des design patterns

Nous ne pouvons réduire un design pattern à un diagramme. La description complète d'un DP utilisera également du langage non-formel et du code.

Le livre *Design Pattern*[GoFDP] présente ses DP en sections selon ce canevas :

- Noms de modèle et classification
- Intention
- Alias
- Motivation
- Indications d'utilisation
- Structure
- Constituants
- Collaborations
- Conséquences
- Implémentation
- Exemples de code
- Utilisations remarquables
- Modèles apparentés

C'est sans oublier la description complète d'un design pattern que nous nous concentrons dans ce texte sur la structure, définie comme une représentation graphique du modèle sous forme d'un diagramme de classes.

En effet, notre étude concerne la mise au point d'un diagramme de classes d'un système à l'aide des DP. C'est ce diagramme qui est notre référence principale : l'alpha et l'oméga de notre processus. Avec la seule connaissance de la structure, le concepteur serait incapable de l'utiliser à bon escient dans son propre diagramme. Ce diagramme de structure résume le DP mais ne le définit pas. D'ailleurs, à bien des égards, il n'est qu'un exemple – certes, le plus évident – d'utilisation du DP. Pourtant, le premier résultat de l'utilisation d'un DP dans une conception de logiciel est bien de voir apparaître sa structure dans le diagramme de classes de ce logiciel.

2.2.3 Un exemple : le design pattern Décorateur

Nom de modèle

Le nom du modèle étudié est Décorateur.

Décorateur[8] (Decorator) est un modèle classé parmi les DP structuraux[9] ; il va nous aider à illustrer les notions théoriques qui précèdent.

[8] Ce design pattern est présenté en détail dans le livre *Design Pattern*[GoFDP], p 203.

[9] Le GoF propose un classement de ses design patterns en trois catégories selon le type de rôle de ces modèles : créateur, structurel ou de comportement. "Les modèles créateurs concernent le processus de création d'objets. Les modèles structuraux s'occupent de la composition de classes ou d'objets. Les modèles de comportement spécifient les façons d'interagir de classes et d'objets et de se répartir les responsabilités."[GoFDP, p. 13]

Problème

Posons le problème de la manière suivante : que faire si nous souhaitons ajouter des caractéristiques et des possibilités multiples à une classe d'objets ? La procédure classique est d'hériter de la classe pour en faire une classe spécifique avec les ajouts souhaités.

Toutefois, *quid* si le concepteur souhaite donner à l'objet plusieurs nouvelles responsabilités[10] différentes, mais dont les caractéristiques pourraient également se combiner ? La conception devrait alors prévoir non seulement les héritages pour chaque nouvelle responsabilité, mais aussi pour chaque combinaison de ces responsabilités. Le diagramme de classes (et par conséquent le programme) serait alors surchargé de manière exponentielle. Ajouter une nouvelle responsabilité signifierait multiplier par deux le nombre de classes spécialisées.

Pour illustrer ce problème de manière concrète, partons de la figure 2.1. Nous souhaitons, à titre d'exemple, ajouter des responsabilités à la classe PERSONNE. Nous voulons avoir des personnes "polyglottes" qui auront la possibilité de traduire les langues, des personnes "mécaniciennes" qui pourront réparer les machines et des personnes "costaudes" qui pourront soulever de lourdes charges. Si, de plus, nous souhaitions pouvoir combiner ces qualités et disposer de PERSONNES qui soient à la fois "polyglottes" et "mécaniciennes", "mécaniciennes" et "costaudes", "polyglottes" et "costaudes" ou les trois à la fois, nous aurions besoin non pas de trois, mais bien de sept spécialisations. Ceci est représenté dans la figure 2.2. Si le diagramme n'avait pas disposé de la classe PERSONNE et que les nouvelles responsabilités avaient dû être ajoutées aux différents types de personnes que sont PASSAGER et MARIN, le nombre de classes supplémentaires aurait été multiplié par deux pour quatorze classes spécialisées. Ajouter un grand nombre de responsabilités provoquerait ainsi une explosion du nombre de sous-classes.

De plus, avec l'héritage, comment transformer une classe déjà instanciée à laquelle on voudrait ajouter des caractéristiques ? Le programmeur devrait probablement créer une nouvelle instance d'une classe mieux spécialisée puis y copier tout ce que l'ancienne classe connaissait et s'en débarrasser. Ce n'est pas très commode ; nous souhaiterions, dès lors, ajouter dynamiquement des responsabilités à une instance.

De même, nous souhaiterions pouvoir retirer dynamiquement les responsabilités d'un objet.

Solution

Une solution nous est fournie par DÉCORATEUR. Ce DP sert à attacher dynamiquement de nouvelles responsabilités à une instance d'une classe d'une

[10] Par exemple, si nous disposions d'une classe IMAGE qui affiche un dessin dans un rectangle de taille prédéfinie à l'écran, nous pourrions souhaiter y ajouter la responsabilité de montrer une barre de défilement lorsque le dessin que cette classe affiche est plus grand que les dimensions de départ ; nous pourrions aussi souhaiter y ajouter la responsabilité de s'encadrer d'un bord de couleur ou encore celle d'afficher une légende lorsque le curseur de la souris est dans le rectangle d'affichage. Nous aurions alors trois responsabilités à ajouter à IMAGE : une barre de défilement, un bord de couleur et une légende.

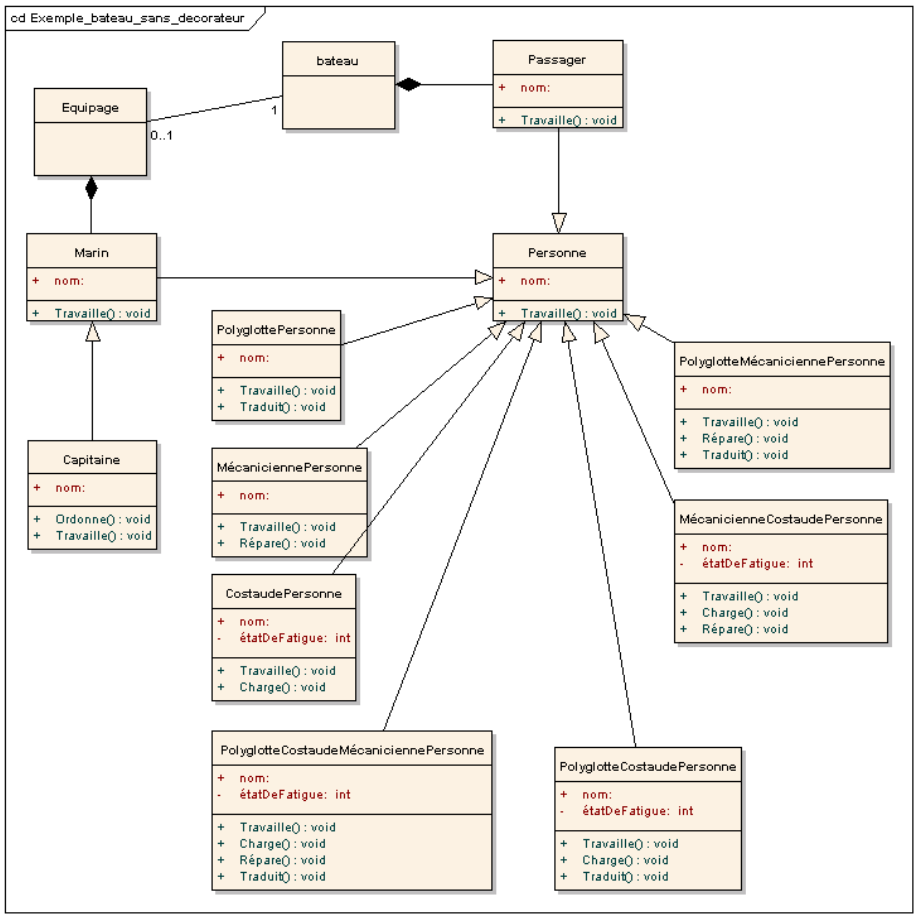

FIG. 2.2: Exemple d'ajout de responsabilités à une classe sans "Décorateur"

manière alternative à l'héritage.

DÉCORATEUR est défini par la structure de la figure 2.3, ainsi que la façon de l'utiliser. Pour utiliser un DÉCORATEUR, le concepteur doit définir une classe abstraite COMPOSANT (Component) qui définit l'interface des objets pouvant être décorés. Il doit définir une ou plusieurs classes COMPOSANTS CONCRETS (Concrete Component) auxquelles pourront être ajoutées des responsabilités. Il doit définir une classe abstraite DÉCORATEUR (Decorator) qui gère une référence vers un COMPOSANT et définit une interface comme COMPOSANT. Enfin, il doit définir un ou plusieurs DÉCORATEURS CONCRETS qui ajoutent les responsabilités désirées aux composants.

Ajouter une responsabilité (c'est-à-dire décorer un COMPOSANT) consiste à instancier un décorateur concret et à lui donner une référence vers le COMPOSANT qu'il décore.

Ainsi, puisque tout COMPOSANT peut être décoré et que les DÉCORATEURS sont des COMPOSANTS (par héritage), on peut décorer un décorateur et donc ajouter successivement autant de responsabilités que nécessaire.

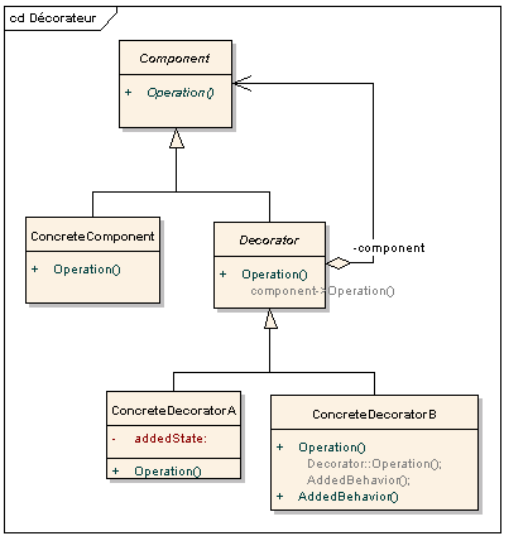

FIG. 2.3: Structure du design pattern Décorateur

Pour le client (l'utilisateur du COMPOSANT), l'instance utilisée est toujours un COMPOSANT et il pourra toujours appeler les opérations définies dans l'interface COMPOSANT.

Si un COMPOSANT est un DÉCORATEUR CONCRET et qu'on appelle une de ses opérations héritée de composant, ce DÉCORATEUR doit appeler à son tour l'opération du COMPOSANT dont il connaît la référence. Eventuellement, il peut avant et après cet appel ajouter des opérations supplémentaires. Appeler l'opération de son COMPOSANT, s'il y a plusieurs DÉCORATEURS imbriqués, transmet l'appel en cascade vers le premier composant qui est normalement un COMPOSANT CONCRET.

Conséquences

En conséquence, utiliser le DÉCORATEUR offre plus de souplesse qu'un héritage statique (on peut dynamiquement inventer toutes les combinaisons, ajouter ou retirer le décorateur). De plus, il permet de définir une classe COMPOSANT (et donc COMPOSANT CONCRET) plus simple en définissant ses caractéristiques au fur et à mesure via les décorateurs. Toutefois, lors de l'utilisation, il faudra prendre garde au fait que le décorateur (DÉCORATEUR CONCRET) et ses composants ne sont pas identiques même s'ils sont les spécialisations d'une même classe. Donc, l'utilisation ne pourra pas tenir compte de l'identité d'un COMPOSANT puisqu'il pourrait très bien être un composant décoré. Enfin, une conception par ce design pattern peut déboucher sur un système composé d'une multitude de petits objets, tous d'apparence voisine. L'apprentissage et les corrections d'un tel système peuvent être ardus.

2.2.4 Enrichissement d'un diagramme de classes

Un concepteur cherchant à développer la structure de son système doit répondre à une série de contraintes fonctionnelles (ce que doit faire le programme) et de contraintes non fonctionnelles (des qualités annexes aussi variées que le temps de réponse ou la flexibilité de la solution). Il peut utiliser les design patterns pour résoudre son problème en procédant, de façon schématique, selon la méthode décrite ci-dessous.

Le concepteur connaît son domaine et il peut d'emblée l'exprimer sous la forme d'un diagramme de classes primitif. Ensuite, il cherche des solutions à ses contraintes notamment parmi les design patterns. Finalement, il combine des concepts propres à son problème avec des design patterns. Le résultat en sera un diagramme de classes spécifique qui sera le modèle de la structure de son système.

Nous appelons l'enrichissement d'un diagramme de classe à l'aide de design patterns, la combinaison d'un diagramme existant avec un design pattern. Soit un diagramme existant qu'on souhaite enrichir, appelons-le *diagramme de base* et soit le diagramme de structure d'un design pattern, appelons-le *diagramme d'enrichissement.* La combinaison des deux diagrammes s'effectue d'une part par ajouts d'éléments nouveaux du *diagramme d'enrichissement* (classes, opérations,...), éventuellement renommés, dans le *diagramme de base ;* et d'autre part, par identifications d'éléments existants du *diagramme de base* avec des éléments du *diagramme d'enrichissement.* En toute généralité, un enrichissement procèdera à la fois d'ajouts de certains éléments et des identifications des autres.

Plus précisément, nous entendons par *identification* la reconnaissance en un élément du diagramme de base, un élément du diagramme d'enrichissement. Lors de la combinaison des deux diagrammes, les éléments identifiés du diagramme d'enrichissement ne sont pas ajoutés, mais les autres éléments, ajoutés, eux, s'agencent autour des éléments identifiés du diagramme de base comme autour de leurs correspondants identifiés du diagramme d'enrichissement.

Nous considérons que chaque élément du diagramme de structure d'un design pattern joue un *rôle* dans ce design pattern. Ce rôle peut être défini de façon très différente pour chaque cas particulier par la fonction de cet élément au sein du design pattern, ses droits et ses devoirs. Par exemple, dans le diagramme de structure de Décorateur, les classes ConcreteDecoratorA et ConcreteDecoratorB jouent toutes deux le rôle de décorateurs concrets. A ce titre, elles doivent toujours hériter de la classe Decorator. Elles doivent toujours implémenter son opération Operation en appelant l'opération Operation de la classe de base à laquelle elles peuvent ajouter des opérations supplémentaires. Elles ont pour fonction d'ajouter des responsabilités à un composant.

L'identification consiste à attribuer le rôle d'un élément du design pattern à un des éléments du diagramme de classe initial.

Par enrichissements successifs de son diagramme de classes, le concepteur générera une structure résolvant son problème qui combine les conséquences des design patterns utilisés.

Comme nous l'avons vu en 2.2.2, un design pattern ne s'exprime pas uniquement en terme de diagramme de classes. Souvent, la compréhension du modèle passe par des explications textuelles, du pseudo-code, etc. Le rôle d'un élément n'est pas forcément strictement défini par le diagramme de structure.

Néanmoins, le diagramme de classes est le moyen le plus évident de représenter la solution. Il permet à la fois de la visualiser ; mais il sert aussi de plan de travail pour la programmation[11]. De plus, avec un peu d'expérience et la réutilisation des mêmes concepts (par exemple par l'utilisation régulière des design patterns), le lecteur d'un diagramme de classes y reconnaîtra assez vite l'intention du concepteur.

Pour illustrer notre propos, reprenons l'exemple présenté précedemment avec la figure 2.1 et enrichissons-le d'un DÉCORATEUR. Nous souhaitons que toute PERSONNE puisse être éventuellement qualifiée de "Polyglotte", "Mécanicienne" ou "Costaude" et ainsi lui attribuer de nouvelles responsabilités.

Nous allons alors identifier la classe PERSONNE du diagramme au COMPOSANT de la structure du design pattern. Nous allons identifier les classes PASSAGER et MARIN aux classes de rôle COMPOSANT CONCRET et ajouter des DÉCORATEURS CONCRETS pour chaque qualificatif que nous souhaitons ajouter. La figure 2.4 résulte de l'enrichissement.

Les identifications qui ont mené au schéma final ne sont pas prédéterminées : elles procèdent de choix qui auraient pu être différents. Par exemple, nous aurions pu ajouter une classe COMPOSANT dont PERSONNE aurait été le composant concret.

Nous avons créé une structure qui permet d'ajouter à toute PERSONNE de nouvelles responsabilités. Dans une implémentation de cette structure, le programme pourrait décorer une instance de PASSAGER de COSTAUDDECORATOR et manipuler l'instance de COSTAUDDECORATOR comme une instance de PERSONNE. Lorsque son opération TRAVAILLE serait appelée, l'instance appellerait l'opération TRAVAILLE de son composant (PASSAGER) puis elle réaliserait sa responsabilité ajoutée, c'est-à-dire, comme le suggère le pseudo-code ajouté dans la figure 2.4, que si son attribut ÉTATDEFATIGUE était sous 5, elle appellerait l'opération CHARGE et incrémenterait son attribut ÉTATDEFATIGUE d'une unité.

2.3 La question

Le processus d'enrichissement d'un diagramme de classes à l'aide de design patterns est au coeur de notre sujet.

> "Des outils CASE[12] proposent maintenant ces design patterns sous forme de librairies de diagrammes de classes que l'on peut réutiliser

[11] D'ailleurs bien des outils proposent de générer du code à partir du diagramme de classes. Il ne s'agit la plupart du temps que d'un canevas à remplir, mais utiliser ce moyen assure une grande adéquation entre le modèle et la réalisation.

[12] "CASE tool" ou "outil CASE" comprend l'acronyme de Computer-Aided Software Engineering. "Un CASE tool est un produit informatique visant au support d'une ou plusieurs activités de génie logiciel dans un processus de développement logiciel." Carnagie Melon Soft-

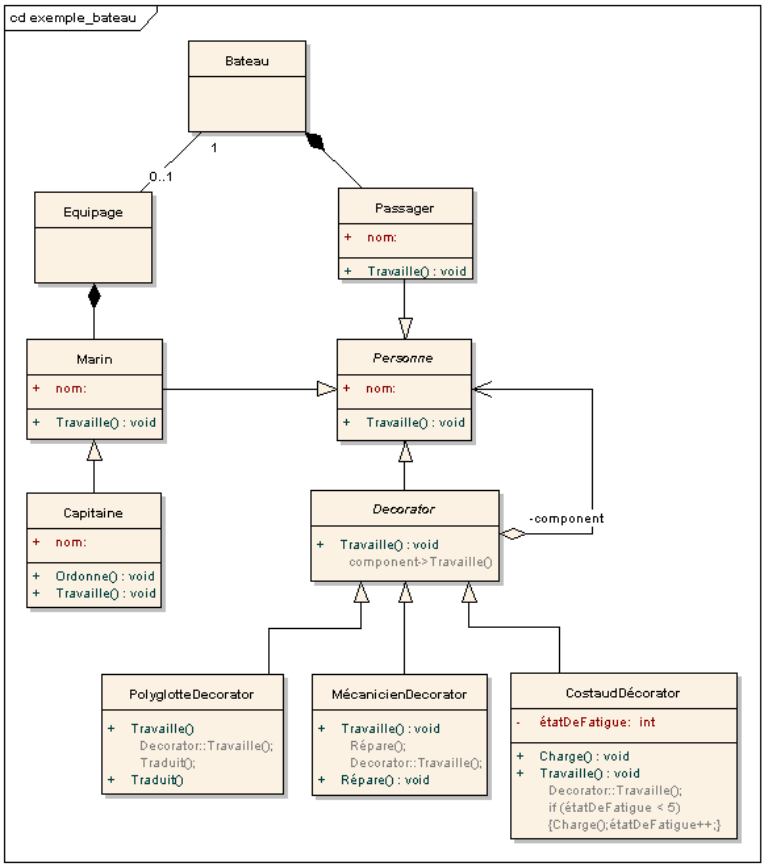

FIG. 2.4: Exemple d'enrichissement par un Décorateur

> dans d'autres diagrammes de classes. Néanmoins, l'apport de ces outils reste limité et donc insuffisant."[13]

Nous proposons une implémentation de procédures d'enrichissement de diagrammes de classes avec des design patterns en utilisant le langage logique Prolog pour une approche transformationnelle. Cette implémentation s'appuie sur un méta-modèle simplifié des diagrammes de classes ; grâce à cette technique, nous pouvons acquérir la connaissance d'un diagramme particulier sous forme d'une représentation logique[14] de ce diagramme. Notre implémentation comprend un jeu de transformations correspondant chacune à un design pattern. La représentation logique d'un diagramme peut alors être enrichie, éventuellement plusieurs fois, par ces transformations.

---

ware Engineering Institute*[CMSEI]*, "Computer Aided Software Engineering (CASE) Environment" *in* `http://www.sei.cmu.edu/legacy/case/case_whatis.html` (consulté le 4 mai 2006). C'est nous qui notons.

[13] V.Englebert, *Propositions de sujets de mémoire*, année académique 2004-2005.

[14] Par opposition à une représentation graphique.

En outre, DB-Main[15], comme d'autres outils CASE, permet de créer visuellement des diagrammes de classes[16]. Notre implémentation confie à DB-Main la représentation visuelle du diagramme ainsi que les interactions avec l'utilisateur via un programme écrit en Voyager 2[17], le langage associé à DB-Main.

[15]DB-MAIN, A product of the LIBD Laboratory, Database Application Engineering, Institut d'Informatique, Rue Grandgagnage, 21 - B-5000 Namur, Belgium, Distributed by REVER S.A., Boulevard Tirou, 130 - B-6000 Charleroi Belgium, `http://www.db-main.be`

[16]Toutefois, DB-Main est conçu avant tout pour l'ingénierie de base de données sur la base de schémas entité-relation. Ces schémas et les diagrammes de classes se correspondent partiellement et DB-Main propose une option de transformation automatique d'un type de schéma vers l'autre. Notre approche simplifiée du méta-modèle des diagrammes de classes nous permet d'utiliser DB-Main avec assez peu de restrictions sur les diagrammes visualisables.

[17]"Voyager 2 est un langage impératif avec des caractéristiques originales telles le type primitif liste avec garbage collection et des requêtes déclaratives sur le référentiel (*repository*) prédéfini de l'outil DB-Main."[V2, p. 3]

Deuxième partie

Etat de l'art

Chapitre 3

Étude de logiciels existants

De nombreux outils CASE proposent de soutenir la création de diagrammes UML et en particulier de diagrammes de classes. Certains de ces outils permettent en outre d'enrichir les diagrammes de classes à l'aide de design patterns.

Les méthodes d'enrichissement de ces logiciels sont souvent similaires et présentent, outre d'indéniables intérêts, de récurrentes lacunes.

Au regard de la quantité d'outils existant, notre étude ne pouvait être exhaustive ; bien d'autres logiciels[1] auraient mérité notre attention, mais le temps disponible et les contraintes du genre nous ont forcé à choisir. Par conséquent, avec l'intention de mettre en valeur certains comportements similaires, nous avons selectionné et étudié une palette de trois logiciels que nous espérons représentatifs :

- DPA Toolkit : *Design Pattern Automation Toolkit* version 0.19.0[2], (`http://dpatoolkit.sourceforge.net`) .
  Nous avons choisi DPA Toolkit, outre pour sa gratuité, parce que son but annoncé – dans le nom même – est l'incorporation de DP dans des diagrammes de classes.
- Enterprise Architect : *Enterprise Architect* version 4.50.747, auteur : Geoffrey Sparks Copyright © Sparx Systems 1998-2005 (`http://www.sparxsystems.com.au/products/ea.html`).
  Enterprise Architect était un logiciel à notre disponibilité immédiate – nous en possédons une expérience professionnelle – ce qui a motivé notre choix.
- Borland Together : *Borland Together Designer 2005, pour Microsoft Visual Studio .NET* Version 2005 SP2 (`http://www.borland.com/us/products/together/index.html`)
  Nous nous devions de choisir Borland Together car il est un des logiciels de modélisation UML les plus connus.

[1] Nous aurions pu étudier par exemple le célèbre IBM Rational Rose et Rational XDE (`http://www-306.ibm.com/software/awdtools/developer/rosexde/`). Un lecteur interressé pourrait consulter la liste d'outils de modélisation UML `http://modelbased.net/uml_tools.html`(consulté le 22 avril 2006) et plus encore `http://modelbased.net/uml_tools.html`(consulté le 22 avril 2006).

[2] La version actuelle est 0.23.0 mais les changements effectués depuis la version 0.19.0 ne concernent pas notre propos.

Les outils que nous critiquons proposent tous un ensemble de fonctionnalités liées à la modélisation telles qu'ingénierie inverse (*reverse engineering*), génération de code ou création et visualisation d'autres types de diagrammes UML[3]. Ces possibilités intéressantes en elles-mêmes ne sont pas discutées ici car elles ne concernent pas notre propos.

Nous présentons donc brièvement pour chaque logiciel étudié la méthode d'intégration de design patterns à un diagramme de classes. Puis, nous tentons de mettre en avant les avantages et les inconvénients de ces intégrations.

3.1 DPA Toolkit

3.1.1 Présentation

DPA Toolkit, pour *Design Pattern Automation Toolkit*, est un outil assez simple conçu pour permettre d'enrichir un diagramme de classes en y ajoutant un DP. La figure 3.1 montre l'interface utilisateur de cet outil CASE.

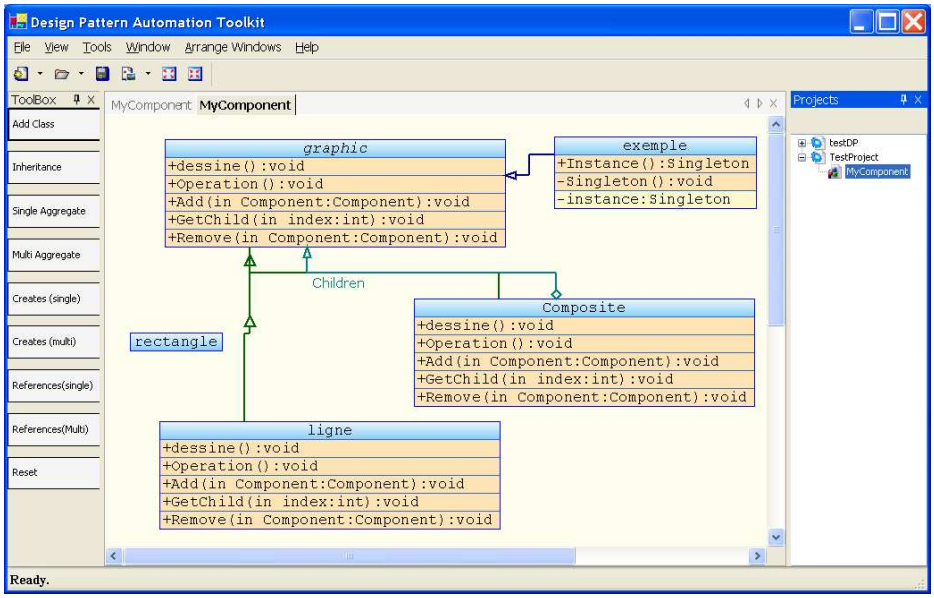

FIG. 3.1: DPA Toolkit – vue globale

Pour ce faire, l'utilisateur doit commencer par créer[4] un diagramme de classes (qui peut éventuellement être vide). Ensuite, il sélectionne un DP dans la liste afin de l'intégrer à son diagramme. L'utilisateur peut alors procéder à l'identification [5] des classes du DP et de son schéma, c'est-à-dire choisir en regard

[3]Diagramme d'activités, diagramme de composants, diagramme de cas d'utilisations, etc.

[4]L'utlisateur crée un "Project" au sein duquel il définit des "Components" lesquels peuvent s'éditer sous forme de diagrammes de classes. Ensuite, il peut ajouter des éléments au diagramme via les boutons de la ToolBox. Ces éléments peuvent être édités pour ajouter, supprimer et modifier leurs propriétés.

[5]Telle que définie en 2.2.4.

des classes du diagramme de structure du DP, des classes de son diagramme de base. Ce processus est illustré par la figure 3.2 où la classe GRAPHIC est identifiée à la classe COMPONENT d'un DP COMPOSITE[6].

Lors de l'intégration du DP, les classes du diagramme de structure du DP qui n'ont pas été identifiées sont simplement ajoutées et les classes qui ont été identifiées sont intégrées en modifiant les classes existantes dans le diagramme de telle manière que les attributs et les opérations de la classe du DP y soient ajoutés. Les associations et les généralisations sont ajoutées également de façon cohérente[7] entre les classes identifiées ou non.

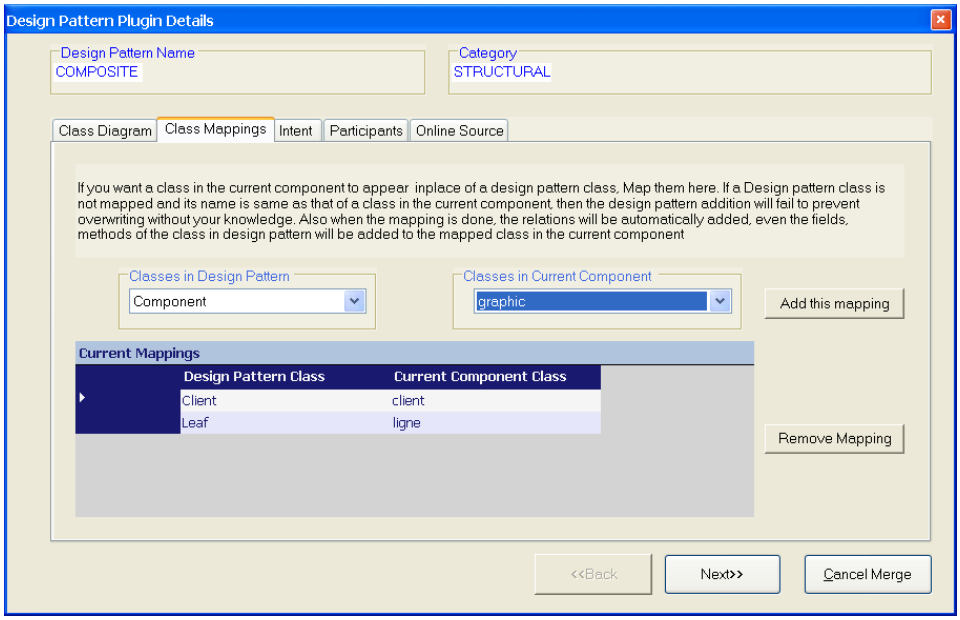

FIG. 3.2: DPA Toolkit – intégration d'un design pattern

3.1.2 Avantages

DPA Toolkit donne une méta-information à propos du DP intégré. Lors de l'intégration, l'utilisateur peut lire des explications concernant l'intention du DP et le rôle de chaque classe dans celui-ci (cfr 3.3). Il peut aussi trouver un lien vers une page web avec une explication détaillée du DP.

De plus, une fois intégrée, chaque classe voit mise à jour sa propriété "summary"[8] qui, outre le nom du rôle de la classe dans le DP, contient une indication

[6] Le DP COMPOSITE est décrit *in Design Patterns* [GoFDP, p. 189].

[7] c'est-à-dire en respectant les rôles des éléments du diagramme de structure du DP.

[8] Voici un exemple de propriété "summary" pour une classe appelée MYCLIENT ayant été identifiée d'abord à la classe SINGLETON du DP homonyme (Le DP SINGLETON est décrit *in Design Patterns* [GoFDP, p. 149].) et identifiée ensuite à la classe CLIENT du DP COMPOSITE :

```
« Summary of MyClass
This class was mapped with Singleton of the design pattern :SINGLETON
```

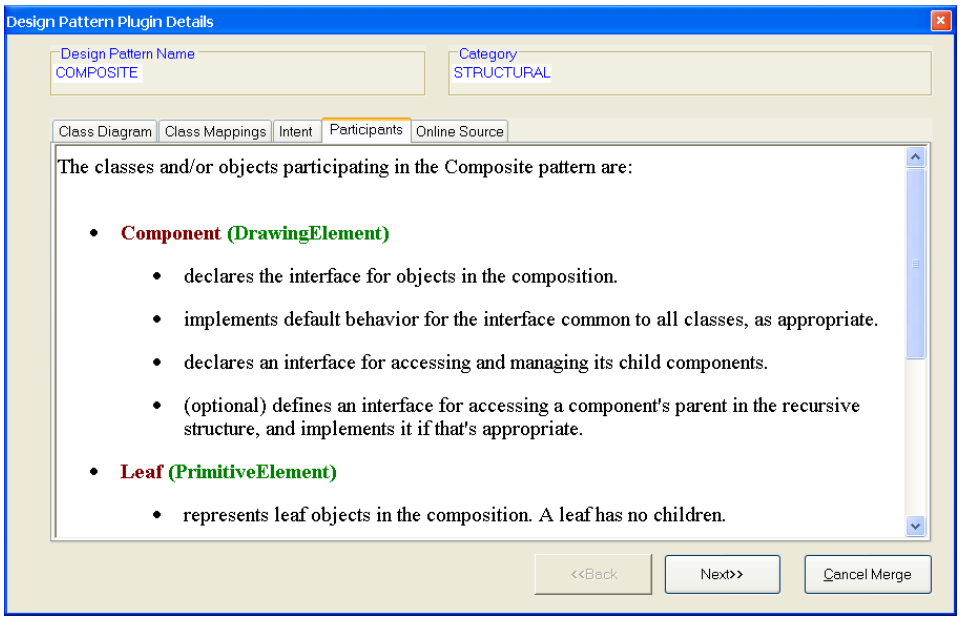

FIG. 3.3: DPA Toolkit – méta-information.

de chaque identification à une classe dans les enrichissements successifs par DP. L'utilisateur peut donc suivre l'historique des intégrations de DP dans le diagramme.

Enfin, DPA Toolkit permet de pouvoir ajouter un DP dans la liste des DP intégrables grâce à un éditeur spécifique. L'utilisateur doit à cet effet y créer un diagramme de classes et écrire la documentation à propos du nouveau DP.

3.1.3 Inconvénients

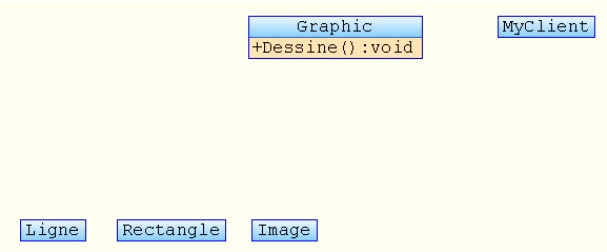

FIG. 3.4: DPA Toolkit – inconvénients – diagramme de base

Malgré ces atouts, DPA Toolkit présente quelques inconvénients :

This class was mapped with Client of the design pattern :COMPOSITE »

1. L'intégration manque de souplesse ; l'utilisateur n'a pas de contrôle sur la multiplicité des éléments qu'il intègre. Tantôt, il est obligé d'intégrer un élément non nécessaire ; tantôt, intégrer plusieurs fois un élément souhaité multiple lui est impossible.
   En effet, bien des DP définissent un type de classes (ou constituant) particulier qui décrit les responsabilités d'un ensemble d'une ou plusieurs classes dont le nombre est variable selon le cas particulier d'utilisation du DP.
   Ainsi, les FEUILLES dans un COMPOSITE peuvent être au nombre de une ou plusieurs classes – autant que nécessaire dans la conception qui nous occupe – de même que les FABRIQUES CONCRÈTES et les PRODUITS ABSTRAITS dans FABRIQUE ABSTRAITE [9] ou encore les DÉCORATEURS CONCRETS dans DÉCORATEURS .
   Avec DPA Toolkit, le nombre d'éléments à ajouter ou identifier est exactement celui du diagramme de structure du DP. Pourtant, comme nous l'avons déjà mentionné en 2.2.2, ce diagramme n'est jamais qu'un exemple d'utilisation du DP. L'utilisateur de DPA Toolkit doit pourtant insérer le même nombre de classes d'un type que celui montré dans son diagramme d'enrichissement alors qu'en toute généralité, leur nombre dépend du contexte particulier de son travail.
   Par exemple, un utilisateur voudrait pouvoir enrichir son diagramme de base d'un DP COMPOSITE avec plusieurs classes de type FEUILLES ; DPA Toolkit ne le permet pas car le diagramme d'enrichissement ne contient qu'une seule feuille et par conséquent, l'intégration de ce DP implique de n'insérer qu'une et une seule feuille. Les figures 3.4 et 3.5 montrent notamment le résultat d'une tentative d'enrichissement d'un diagramme par un COMPOSITE avec trois feuilles (les classes LIGNE, RECTANGLE et IMAGE). L'utilisateur a dû choisir une des trois classes qu'il souhaitait intégrer comme FEUILLES.
   Cette remarque ne s'applique pas qu'aux classes et peut s'étendre aux autres éléments du diagramme tels qu'attributs et opérations des classes.

2. Les identifications ne concernent que des classes ; l'utilisateur ne peut espérer identifier d'autres types d'éléments d'un diagramme de classes tels qu'attribut ou opération.
   Ainsi, les opérations et attributs du diagramme d'enrichissement sont toujours ajoutés même s'ils sont déjà présents.
   A titre d'exemple, considérons à nouveau l'enrichissement par le DP COMPOSITE du diagramme de la figure 3.4 vers celui de la figure 3.5. Dans le DP COMPOSITE, les opérations de la classe COMPOSANT, identifiée à GRAPHIC, doivent être héritées et implémentées dans les classes FEUILLES et COMPOSITE. Ces opérations sont représentées par une opération nommée OPERATION dans le diagramme de structure de ce DP et devraient être identifiées à une ou plusieurs opérations de la classe COMPOSANT choisie – ici, GRAPHIC. *A contrario*, DPA Toolkit ajoute l'opération OPERATION dans les classes COMPOSANT, COMPOSITE et FEUILLES. Heureusement, créer un héritage dans DPA Toolkit copie les attributs et les opérations de la classe générale vers la classe spécialisée. Donc,

[9] Le DP FABRIQUE ABSTRAITE est décrit *in Design Patterns* [GoFDP, p. 101].

dans notre exemple, l'utilisateur corrigera facilement son diagramme en effaçant les opérations OPERATION de son diagramme enrichi.

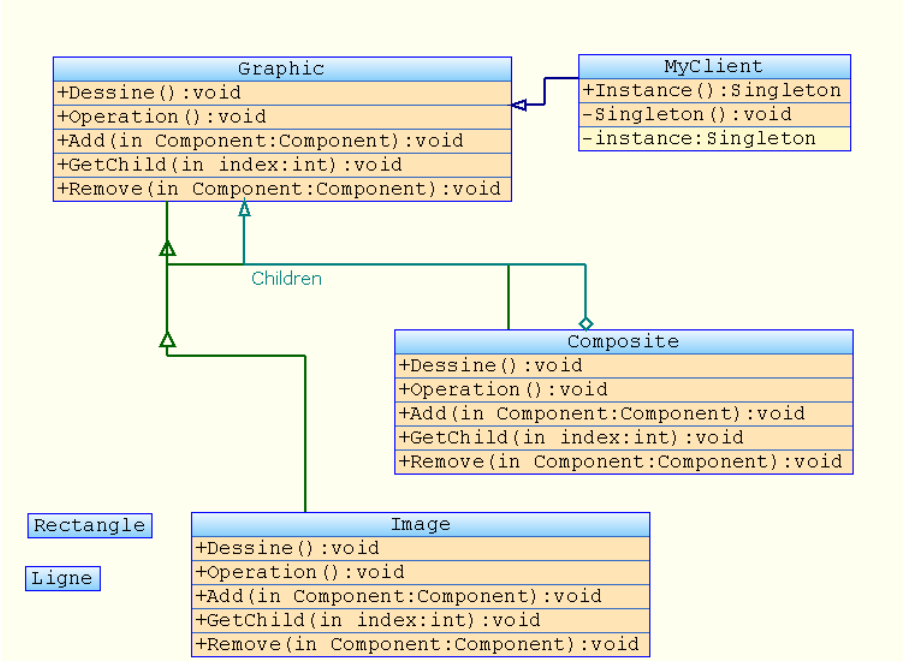

FIG. 3.5: DPA Toolkit – inconvénients – diagramme enrichi

3. Les types des attributs et la signature des opérations ne sont pas modifiés pertinemment et gardent les types qui correspondent au diagramme d'enrichissement.
   Par exemple, si un utilisateur identifie une classe MYCLIENT à la classe SINGLETON du DP SINGLETON, on s'attend à voir une méthode publique INSTANCE renvoyer le type MYCLIENT, mais le résultat sera une méthode INSTANCE de type SINGLETON (quand bien même aucune classe nommée SINGLETON n'existerait dans le diagramme). Ceci est également illustré dans les figures 3.4 et 3.5.
4. L'historique des intégrations de DP est limitée à une note sur chaque classe sous forme d'un texte ajouté au texte de la propriété « summary » des classes. Cette conception de l'historique est pauvre : elle ne fournit ni une manipulation aisée du concept – telle, par exemple, la connaissance simultanée de toutes les classes jouant le même rôle dans un enrichissement –, ni des fonctionnalités dérivées – telles, par exemple, l'annulation d'un enrichissement particulier.
5. L'intégration d'un DP n'est pas contrainte par des préconditions ; rien n'empêche de tenter un enrichissement syntaxiquement faux[10], ni d'enri-

[10]Par exemple, un enrichissement qui ajoute à une classe une opération de nom déjà utilisé dans celle-ci.

chir un diagramme d'un design pattern inadapté[11].
Si en conséquence, un enrichissement devait produire un diagramme syntaxiquement non valide, le processus d'intégration de DPA Toolkit annulerait les ajouts litigieux au fur et à mesure de leur tentative d'insertion. Si par contre, l'erreur d'intégration ne concernait que la logique des DP utilisés, rien ne pourrait l'empêcher.
A titre d'exemple, notons que l'enrichissement par un singleton de la classe MyClient – déjà Singleton – de la figure 3.5 n'est pas interdit. Toutefois, tenter cet enrichissement produira une erreur (illustrée en 3.6) sur chaque tentative d'ajouter un attribut ou une opération du même nom. De plus, la propriété "summary" de la classe sera modifiée et arborera deux fois la mention signalant que la classe a été identifiée au Singleton du DP Singleton.

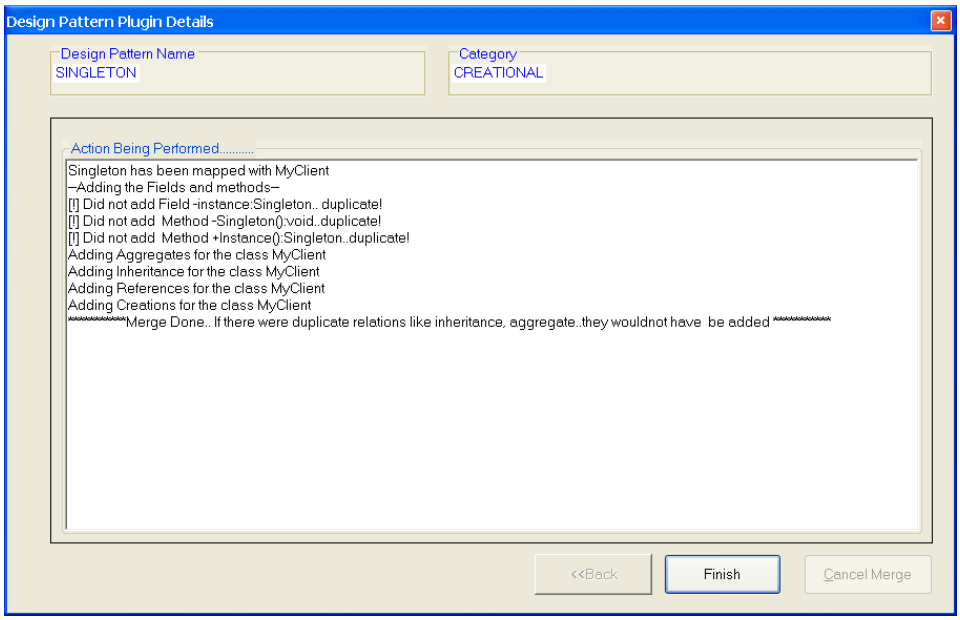

Fig. 3.6: DPA Toolkit – inconvénients – erreur d'intégration

6. Un classe non abstraite identifiée à une classe abstraite d'un diagramme d'enrichissement ne devient pas abstraite. Bien que transformer la classe identifiée en classe abstraite ne soit pas pertinent dans tous les cas de figure, nous pensons que, si l'abstraction n'est pas imposée par défaut, l'attention de l'utilisateur devrait être attirée sur ce point.
   A titre d'illustration, le diagramme de la figure 3.5 présente une classe Graphic concrète, contrairement à Composant du diagramme de structure de Composite auquel elle a été identifiée.
7. Conformément à la sémantique UML, DPA Toolkit interdit d'ajouter des opérations et attributs de même nom au sein d'une classe. Si l'intégration d'un DP doit ajouter un membre et que le nom existe déjà, l'outil CASE

[11] Par exemple, l'identification à un Singleton d'une classe déjà Singleton.

abandonne son insertion, mais sans que l'intégration du DP ait échoué. Ceci peut générer des diagrammes non-conformes à l'esprit du DP inséré. Par exemple, si l'utilisateur tente d'identifier un SINGLETON avec une classe qui contient un attribut public INSTANCE, le SINGLETON créé gardera son attribut INSTANCE avec la visibilité "public" ce qui est contraire à l'idée de base de ce DP.

8. Si l'ajout systématique des opérations d'une classe générale dans une classe spécialisée est syntaxiquement logique, il ne l'est pas toujours selon l'esprit du DP. Ainsi, les opérations ADD, REMOVE et GETCHILD apparaissant dans les feuilles de COMPOSITE comme illustré en 3.5, même si elles font formellement partie de l'héritage de COMPOSANT, ne doivent pas être implémentées comme dans la classe de rôle COMPOSITE[12].
   A cet ajout systématique aurait été préférable un ajout dépendant du rôle de la classe dans le DP particulier.

Si nous entendons ici par transformation, le changement de nature ou de propriétés d'un élément du diagramme[13], nous pouvons conclure que l'intégration d'un DP dans ce logiciel se réduit à l'ajout d'un diagramme de classes particulier avec identification de certaines classes au diagramme de départ et pas à la transformation du diagramme par le DP. En conséquence, si on excepte les cas triviaux, intégrer un DP à un diagramme de classes avec DPA Toolkit demandera toujours quelques corrections manuelles.

3.2 Enterprise Architect

3.2.1 Présentation

Bien qu'Enterprise Architect[14] propose un grand nombre de fonctionnalités annexes – telles que la création et la manipulation d'autres types de diagrammes – ce logiciel ressemble beaucoup à DPA Toolkit spécialement en ce qui concerne l'intégration d'un DP dans un diagramme de classes.

En effet, une fois le diagramme de base créé, l'utilisateur doit choisir le DP à intégrer dans une liste puis éventuellement identifier des classes du diagramme de base à d'autres du diagramme de structure du DP.

Un formulaire permet à l'utilisateur d'entrer les paramètres de l'enrichissement ; il est illustré en 3.8 selon le même scénario que lors de la critique de DPA Toolkit : l'enrichissement d'un diagramme par un COMPOSITE. Dans ce

[12] Au contraire, un traitement spécifique pourrait leur être attribué. Donc, si le diagramme ne les ignore pas, leurs différences par rapport à l'implémentation dans la classe de rôle COMPOSITE devraient être signalées afin d'éviter une mauvaise interprétation. En l'occurence, une implémentation possible peut se réaliser en considérant qu'une classe FEUILLE est un COMPOSANT qui n'a jamais d'ENFANT et donc que GETCHILD dans une FEUILLE ne renvoie jamais aucun ENFANT. Le lecteur pourra trouver une explication plus détaillée dans la partie "Implémentation" du modèle COMPOSITE *in Design Pattern* [GoFDP, p. 194].

[13] Par exemple, changer le stéréotype ou la visibilité d'un membre d'une classe serait qualifié de transformation.

[14] La figure 3.7 montre l'interface visuelle du produit.

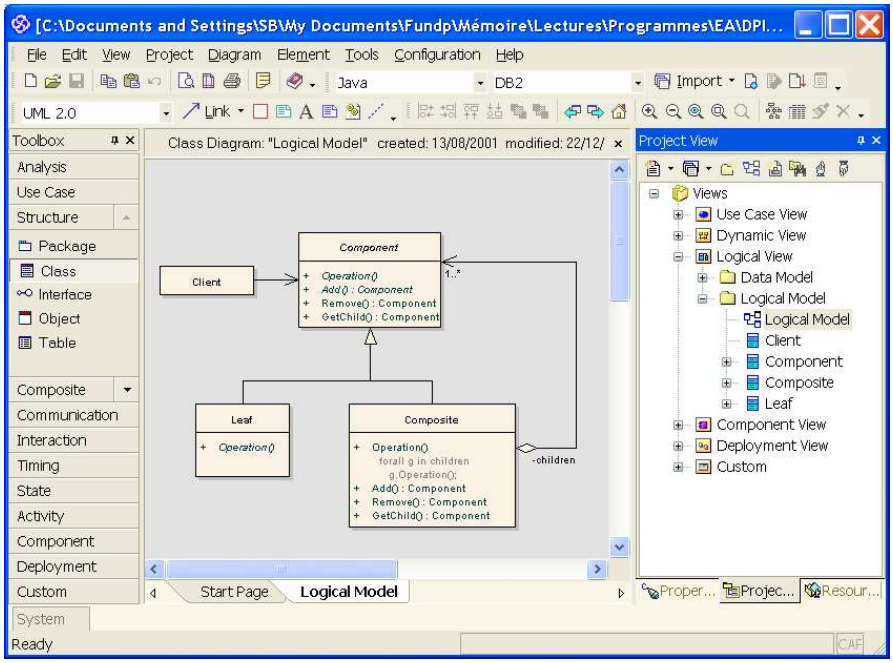

FIG. 3.7: Enterprise Architect – vue globale

formulaire, chaque classe du diagramme d'enrichissement est qualifié d'une action "create" ou "merge" dont "create" est l'action par défaut. Choisir l'action "create" provoque l'ajout de la classe lors de l'enrichissement tandis que choisir l'action "merge" correspond à une identification. Une classe d'action "create" pourra voir modifié le nom de la classe à insérer tandis qu'une classe d'action "merge" devra être identifiée à une classe du diagramme de base.

L'intégration du DP est semblable à celle de DPA Toolkit : les classes d'action "create" sont simplement ajoutées et les classes d'action "merge" sont intégrées en modifiant les classes existantes dans le diagramme de la façon suivante : les membres de la classe identifiée du diagramme de structure sont ajoutés à la classe identifiée du diagramme de base. Les associations et les généralisations sont ajoutées également entre les classes identifiées ou non du DP.

3.2.2 Avantages

Comme DPA Toolkit, Enterprise Architect fournit une méta-information sous la forme d'une description du DP et d'une "element note" pour chaque classe du diagramme d'enrichissement qui explique le rôle de cette classe dans le DP.[15] De plus, lors de l'enrichissement, Enterprise Architect ajoute une note spéciale pour quelques opérations sous la forme d'une propriété de l'opération appelée

[15] Exemple de note pour un Singleton : "This class (a) defines an Instance operation that lets clients access its unique instance, and (b) may be responsible for creating its own unique instance."

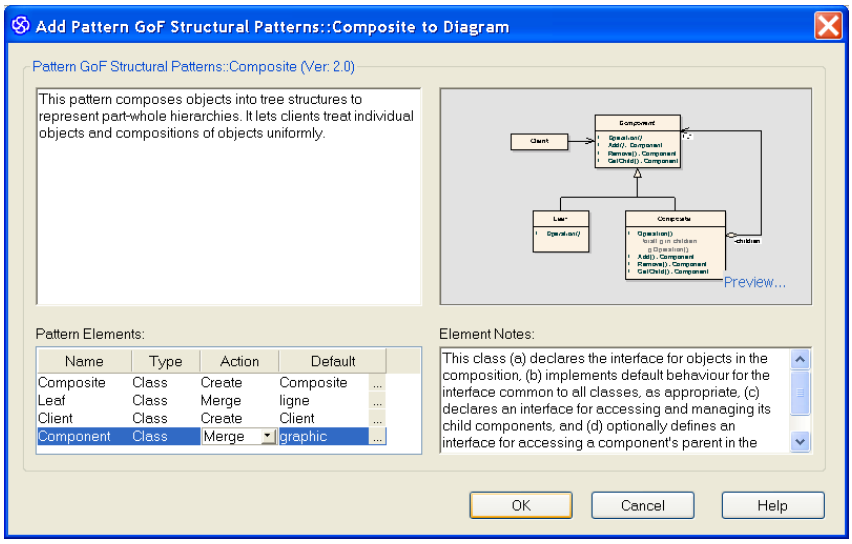

FIG. 3.8: Enterprise Architect – intégration d'un design pattern

"behavior"[16]. Cette propriété met en valeur certains points cruciaux du DP à ne pas perdre de vue lors de l'implémentation[17].

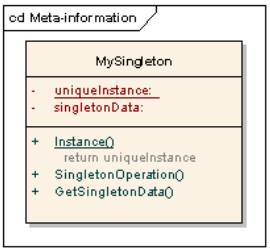

FIG. 3.9: Enterprise Architect – méta-information

Chaque classe a une propriété "note" (qui n'est pas une "note" habituelle de l'UML). Les notes des classes issues de l'intégration d'un DP – qu'elles aient été ajoutées ou identifiées – ont pour texte celui de l'"element note" du diagramme d'enrichissement qui explique sa fonction au sein du DP. Toutefois, contrairement à DPA Toolkit, Enterprise Architect n'ajoute pas les notes des DP successivement intégrés et garde uniquement celles du premier DP. Ceci ne constitue donc pas du tout une notion d'historique comme dans DPA Toolkit.

Enterprise Architect dispose également d'une fonctionnalité d'import de nouveaux DP sur base d'un fichier descriptif (en format XML). Ce fichier peut être

[16]Par exemple, dans SINGLETON, l'opération INSTANCE a le "behavior " suivant : "return uniqueInstance" car UNIQUEINSTANCE est l'attribut qui représente l'instance de la classe si elle existe. Cette propriété est illustrée à la figure 3.9 (en gris sous la méthode INSTANCE).

[17]D'ailleurs, si l'utilisateur emploie la fonctionnalité de génération de code, le "behavior" est écrit en remarque dans le code généré.

généré à partir d'un diagramme de classes créé précédemment. La figure 3.10 présente le formulaire de création d'un nouveau DP.

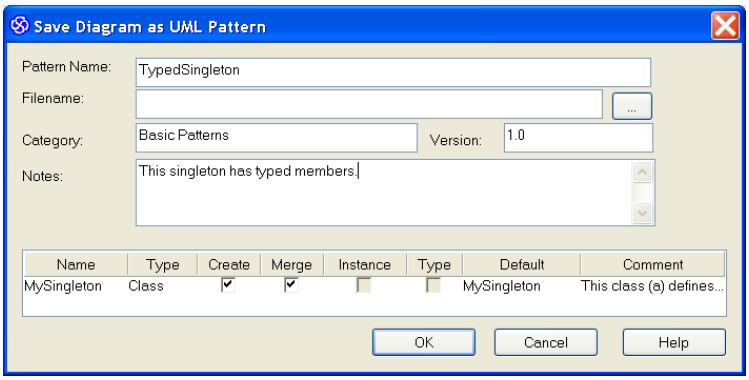

FIG. 3.10: Enterprise Architect – création d'un nouveau DP

Contrairement à DPA Toolkit, Enterprise Architect ne donne pas de type aux membres ajoutés lors d'un enrichissement. Nous pensons que ceci est préférable à l'erreur de typage de DPA Toolkit. De plus, ne pas typer les membres n'est pas inéluctable avec Enterprise Architect. En effet, les membres des diagrammes d'enrichisement y sont non typés, mais si un utilisateur crée un nouveau DP dont un membre a pour type sa propre classe ou celle d'une autre classe de son diagramme et qu'il tente un enrichissement d'un diagramme avec ce nouveau DP, alors Enterprise Architect ne se trompe pas : les membres auront bien les types souhaités, à savoir ceux de la classe créée ou identifiée. Malheureusement, cette opération peut se réaliser avec quelques erreurs [18].

3.2.3 Inconvénients

Si Enterprise Architect est proche de DPA Toolkit sur le plan des avantages, il l'est aussi sur celui des inconvénients :

1. Enterprise Architect manque lui aussi de souplesse concernant le nombre de classes intégrées. Il ajoute ou identifie exactement le nombre de classes définies dans le diagramme de structure.
   Les figures 3.11 et 3.12 montrent l'enrichissement du diagramme de base représenté par la première figure par COMPOSITE et SINGLETON. Dans le résultat final illustré dans la seconde figure et obtenu en identifiant MYCLIENT à SINGLETON dans l'enrichissement par Singleton et, dans l'enrichissement par COMPOSITE, en identifiant IMAGE à la FEUILLE, MYCLIENT à CLIENT et GRAPHIC à la fois à COMPOSITE et COMPOSANT, on peut voir illustrée la situation d'une seule FEUILLE identifiable quand l'utilisateur aurait souhaité en voir trois.

[18] La version testée d'Enterprise Architect (4.50.747) semble manifester des problèmes dans son implémentation qui se présentent lors d'enrichissement à l'aide DP créés par l'utilisateur. Nous avons constaté l'insertion d'une nouvelle classe de même nom plutôt que l'ajout des membres dans la classe identifiée ou l'ajout dans le diagramme de classes issues d'autres diagrammes.

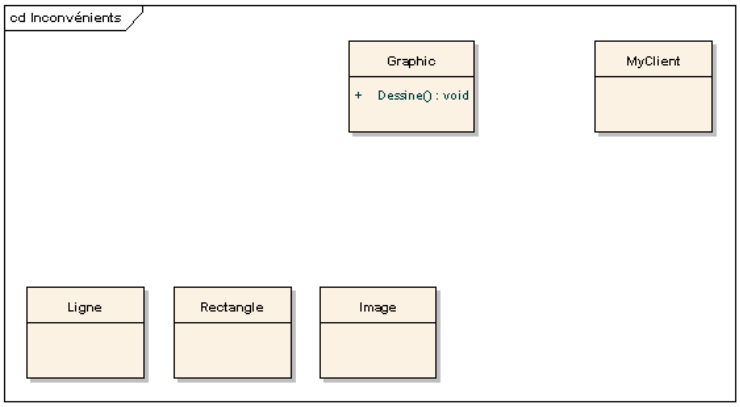

FIG. 3.11: Enterprise Architect – inconvénients – diagramme de base

2. Dans ce logiciel également, les identifications ne concernent que des classes et jamais un autre type d'élément des diagrammes de classes. Pourtant, comme le montre la figure 3.8, le formulaire d'intégration du DP affiche un type d'élément à identifier, mais ce type est toujours "classe". De même, le formulaire de création d'un nouveau DP (voir 3.10) ne permet de créer des éléments à identifier que de type "classe".
   Comme dans DPA Toolkit, les membres d'une classe du diagramme d'enrichissement sont toujours ajoutés à la classe correspondante du diagramme enrichi. Toutefois, Enterprise Architect ne copie pas les membres dans les classes spécialisées lorsqu'un héritage est créé. Donc, après l'intégration d'un DP où un héritage est créé, l'utilisateur devra non seulement supprimer ou renommer les membres ajoutés, mais aussi propager manuellement ces changements dans les classes spécialisées.
   Les figures 3.11 et 3.12 illustrent le problème ici évoqué : l'opération DESSINE n'a pas été propagée, mais l'opération OPERATION (que l'utilisateur devra enlever ou renommer) a été insérée.
3. Enterprise Architect ne propose aucun historique des enrichissements à l'aide de DP.
4. Enterprise Architect ne semble pas appliquer de pré-condition sur les transformations. L'utilisateur peut aussi bien demander des enrichissements syntaxiquement faux que des enrichissements inadaptés aux concepts des DP utilisés. Comme Enterprise Architect ne vérifie pas toutes les règles syntaxiques de l'UML lors de la création des éléments d'un diagramme[19], l'utilisateur pourra créer des diagrammes syntaxiquement faux. Par exemple, lors de l'enrichissement d'un diagramme par le DP COMPOSITE, l'utilisateur peut choisir d'identifier la même classe à la fois comme COMPOSANT et comme COMPOSITE. Le diagramme résultant montre une classe jouant à la fois les rôles de COMPOSANT et de COMPOSITE et héritant d'elle-même, ce qui est une erreur à la fois du point de vue de la syntaxe UML, mais aussi de la logique du DP COMPOSITE. Le diagramme

[19] Il permet notamment de créer plusieurs classes de même nom dans un même diagramme.

de la figure 3.12 montre le résultat d'une telle manipulation sur le diagramme de la figure 3.11 où la classe GRAPHIC a été identifiée à la fois à COMPOSANT et à COMPOSITE.

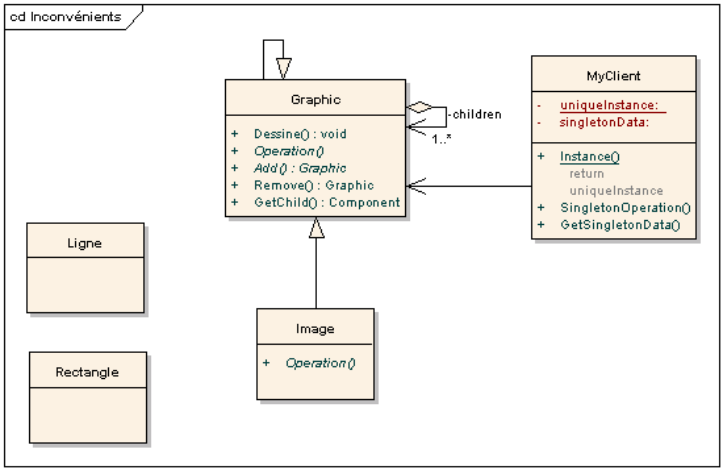

FIG. 3.12: Enterprise Architect – inconvénients – diagramme enrichi

5. Tout comme avec DPA Toolkit, un enrichissement avec Enterprise Architect ne convertit jamais une classe non-abstraite en classe abstraite.
   A titre d'exemple, considérons le diagramme de la figure 3.12 qui montre après enrichissement une classe GRAPHIC non-abstraite alors qu'elle représente la classe COMPOSANT d'un DP COMPOSITE.
6. Tout comme DPA Toolkit, Enterprise Architect ne transforme pas les membres déjà présent lors d'un enrichissement ; il les ignore tout en ajoutant les membres non-présents.
   A titre d'exemple, considérons l'enrichissement du diagramme représenté à la figure 3.13 par l'identification de son unique classe MYCLASS à la classe SINGLETON du DP SINGLETON. Le diagramme de base possède un attribut UNIQUEINSTANCE non-statique et public[20]. Le résultat est fourni par la figure 3.14 ; l'attribut UNIQUEINSTANCE y est toujours non-statique et de visibilité publique ce qui est en désaccord avec le DP Singleton. Transformer UNIQUEINSTANCE en attribut statique et privé ou mieux ajouter un nouvel attribut jouant ce rôle avec un autre nom aurait, nous semble-t-il, été souhaitable ; abandonner l'enrichissement en signalant l'erreur était une alternative.
   L'utilisateur peut donc encore générer des diagrammes qui trahissent l'idée du DP.

En conclusion, de façon fort similaire à DPA Toolkit, Enterprise Architect permet d'enrichir un diagramme de classes par l'ajout d'éléments provenant d'un autre diagramme de classes spécifique, le diagramme de structure d'un

[20] Dans le diagramme, une opération publique est précédée d'un signe "+" et privée d'un signe "-" ; une opération statique est soulignée.

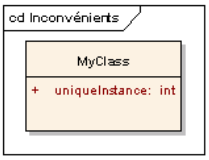

FIG. 3.13: Enterprise Architect – inconvénients – uniqueInstance publique, avant

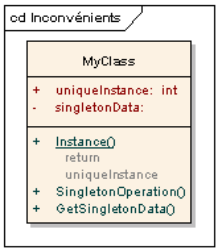

FIG. 3.14: Enterprise Architect – inconvénients – uniqueInstance publique, après

DP. Toutefois, il ne permet pas de transformer une partie du diagramme selon l'esprit du DP. Là encore, un travail manuel sera presque toujours nécessaire pour corriger le diagramme une fois le DP intégré.

3.3 Borland Together

3.3.1 Présentation

Borland Together, logiciel aux nombreuses fonctionnalités, s'intègre dans différents environnements de développement[21]. Pour notre étude, nous avons choisis la version Borland® Together® Designer 2005, pour Microsoft® Visual Studio® .NET qui se présente comme un package d'extension de modélisation UML pour Visual Studio .NET. La figure 3.15 expose une vue globale de l'outil.

Comme les logiciels précédemment étudiés, Borland Together propose une boîte à outils qui permet, une fois le diagramme de classes créé, d'y insérer classes et relations. Cette fois encore, l'utilisateur peut choisir un DP dans une liste et identifier des éléments du diagramme de structure avec celles de son diagramme de base. Le formulaire d'intégration d'un DP est illustré par la figure 3.16.

L'enrichissement d'un diagramme à l'aide de DP dans Borland Together présente de nombreuses différences avec DPA Toolkit ou Enterprise Architect, différences, pour la plupart, à l'avantage du logiciel que va critiquer ce troisième point.

[21] Les éditions couvrant différentes capacités s'intègrent dans trois environnements : *Eclipse*, *Microsoft Visual Studio .Net* et *JBuilder*. D'anciennes versions existent qui s'intègrent dans d'autres environnements ou dans aucun.

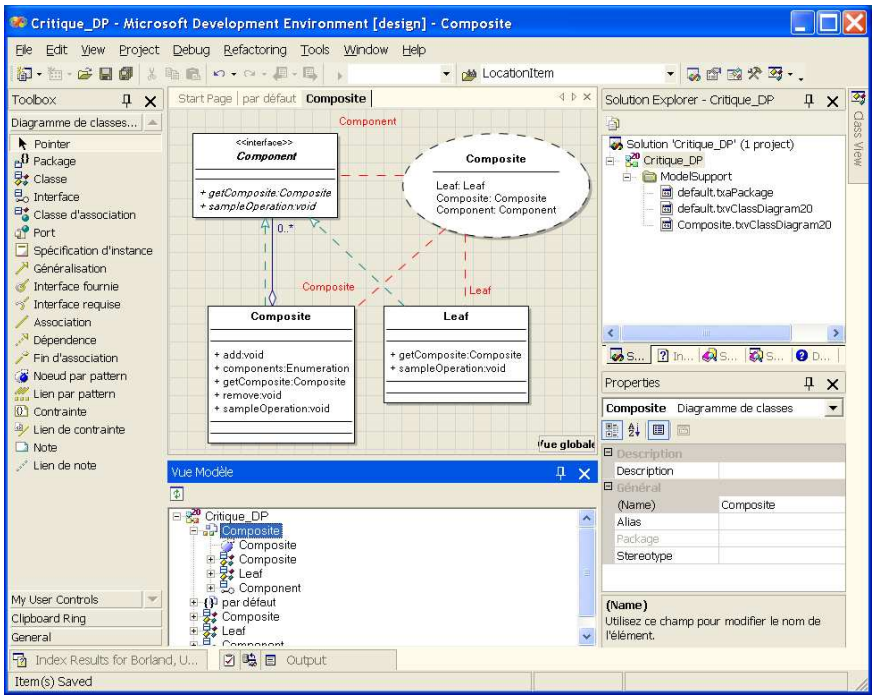

FIG. 3.15: Borland Together – vue globale

3.3.2 Avantages

Tout d'abord, notons les avantages que Borland Together partage avec les autres outils étudiés : la présence de méta-information, la possibilité de création de nouveaux patterns et, comme Enterprise Architect, un typage relatif correct après enrichissement.

En effet, Borland Together donne à lire, pour chaque pattern, une description comprenant les objectifs du DP et le rôle de chaque constituant dans ce même DP accompagné d'une courte explication sur la façon de l'utiliser, comme illustré par la figure 3.16. Cette information dispensée lors de la "paramétrisation" de l'enrichissement n'est plus disponible une fois le DP intégré au diagramme de base.

Toutefois, le rôle au sein du DP de chaque élément ajouté ou identifié est connu à tout moment grâce à l'insertion dans le diagramme d'un nouveau type d'élément "pattern", représenté par un ovale. Cet élément est visible notamment dans la figure 3.15 où il porte le nom COMPOSITE.

Le nouvel élément "pattern" correspond à une note spécifiant chaque enrichissement par un DP. Plus précisément, les vingt-trois "Gof Patterns" [22] sont spécifiés comme "First Class Citizen", une qualité particulière de certains modèles de conception de Borland Together qui leur vaut l'insertion de cet élément particulier.

[22] c'est-à-dire les vingt-trois DP présentés dans le livre *Design Pattern* [GoFDP].

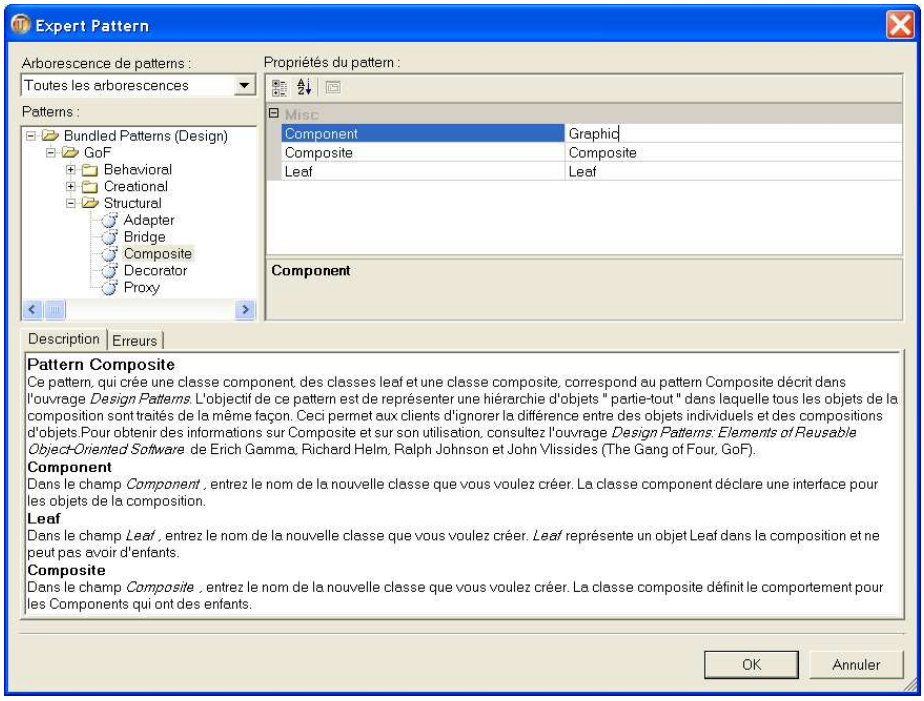

FIG. 3.16: Borland Together – intégration d'un design pattern

Chaque classe participant au DP y est reliée par une ligne pointillée rouge annotée par le nom du rôle que joue la classe dans le DP. Une liste des "participants" du pattern affichée dans l'ovale reprend les classes ajoutées ou identifiées sous la forme du nom de leur rôle dans le DP suivie d'un double point et enfin du nom de la classe.

De plus, via cet élément "pattern", l'utilisateur peut, après l'insertion du DP, ajouter des "participants" par ajout ou identification d'un élément. L'enrichissement peut donc se réaliser en plusieurs phases : la phase d'insertion du DP suivie éventuellement d'une ou plusieurs phase(s) d'ajout d'éléments "participants".

Seuls certains DP et certains rôles permettent cet ajout. Ainsi, l'utilisateur ne pourra rien ajouter à un SINGLETON, mais pourra ajouter une opération COMPOSITE ("Composite method")[23] ou une classe FEUILLE ("leaf") à un "pattern" COMPOSITE. La figure 3.17 illustre l'ajout d'une classe FEUILLE NEWLEAF et d'une opération COMPOSITE NEWCOMPOSITEMETHOD. Notons que, bien que des éléments de type attribut ou opération puissent être ajoutés par ce biais, seuls les éléments de type classe sont repris dans la liste des participants.

L'ajout de nouveaux participants à l'élément "pattern" constitue une avancée par raport aux outils CASE précédemment étudiés en ceci que la souplesse de l'enrichissement est bien plus grande.

[23] C'est-à-dire une nouvelle opération dans les classes COMPOSANT et COMPOSITE qui sera implémentée dans chaque feuille.

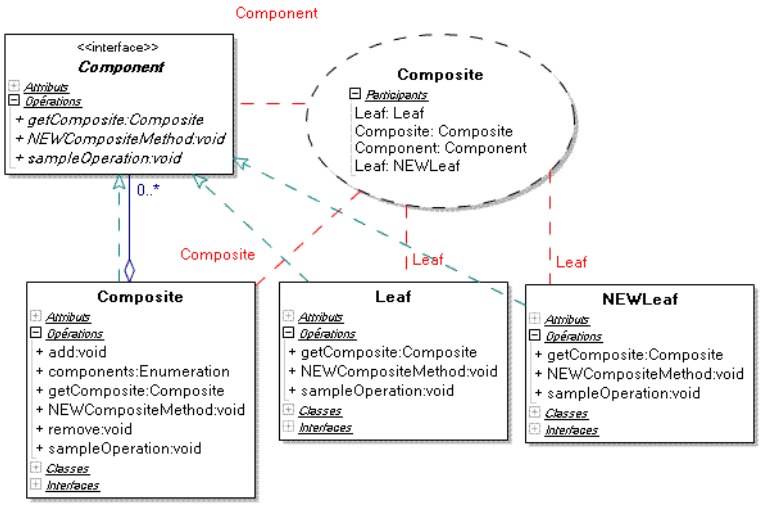

FIG. 3.17: Borland Together – ajout de participants dans un pattern

Pour clore notre propos concernant l'élément "pattern" des diagrammes de classes de Borland Together, notons, d'une part, que cet élément "pattern" n'a pas d'autre sens dans le diagramme de classes que de signifier l'utilisation d'un DP et qu'il ne produit rien lors d'une génération de code ; d'autre part, il ne constitue pas une notion d'historique puisqu'il ne donne pas de renseignement à propos de l'ordre des enrichissements.

Par ailleurs, la création de nouveaux modèles est, encore une fois, basée sur un formulaire et générée à partir d'un diagramme de classes que l'utilisateur aura précédemment conceptualisé. Chaque classe présente dans le diagramme de création représente un rôle dans le modèle (un participant). Le formulaire permet de décider des propriétés des participants du nouveau pattern : nom, nom d'affichage, valeur par défaut (le nom par défaut des classes ajoutées), description et une valeur binaire "utiliser l'existant". Cette valeur, si elle est vraie, permettra d'identifier une classe à ce participant lors de l'utilisation du pattern dans un diagramme.

Lorsqu'un utilisateur enregistre un diagramme comme pattern, il peut choisir de "créer l'objet comme pattern" ce qui fera du modèle un "First Class Citizen" et provoquera l'ajout d'un élément de type "pattern" lors de l'utilisation du nouveau modèle.

Les modèles que l'interface permet de créer sont moins sophistiqués que ceux déjà présents tels que les "GoF Patterns". Ils ne permettent pas d'ajouter des participants après insertion du pattern ni d'identifier d'éléments autres que des classes.

Outre ces avantages, Borland Together permet l'ajout et l'identification d'autres éléments que des classes dans l'enrichissement par certains DP lors de la première phase d'intégration d'un DP, mais aussi par la suite lors des phases

d'ajout d'éléments "participants". Ainsi, SINGLETON ou STRATÉGIE[24] comptent des opérations dans la liste des éléments identifiables lors de la phase d'intégration de l'enrichissement et même lors de la phase d'ajout d'éléments dans la cas de STRATÉGIE tandis que COMPOSITE n'en compte que lors de la phase d'ajout.

3.3.3 Inconvénients

Ces nombreux avantages n'empêchent pas quelques inconvénients :

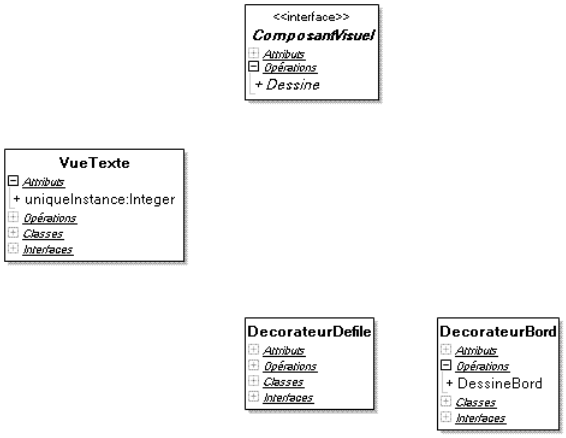

FIG. 3.18: Borland Together – inconvénients – diagramme de base

1. Si la souplesse de Borland Together quant au nombre d'éléments intégrés ou non peut-être considérée comme assez bonne grâce aux ajouts d'éléments après intégration, elle est toujours limitée lors de la première phase de l'enrichissement où un et un seul élément d'un rôle donné peut et doit être intégré.
   De plus, certains attributs et opérations sont toujours systématiquement ajoutés sans qu'on puisse les identifier ou changer leur nom par défaut au moment de l'intégration. Ces éléments du diagramme de structure semblent n'avoir été reconnus ni comme des éléments spécifiques pouvant être identifiés ou ajoutés, ni comme des participants pouvant être ajoutés par la suite.
   Enfin, les opérations et les attributs préexistants dans une classe ne sont pas propagés dans les nouvelles classes qui la spécialiseraient lors d'une intégration. L'utilisateur doit donc, afin de finaliser son enrichissement, non seulement supprimer des opérations non nécessaires, mais aussi ajouter par la suite des opérations nécessaires.[25]

[24] Le DP STRATÉGIE est décrit *in Design Patterns* [GoFDP, p. 369].

[25] Comme une identification d'opération ne permet pas de choisir le type de retour (ou une identification d'attribut le type) et que les membres ne sont pas propagés d'une classe générale vers une classe spécialisée, l'utilisateur peut souhaiter changer le type de retour d'une

A titre d'exemple, considérons l'enrichissement du diagramme présenté à la figure 3.18 successivement par DÉCORATEUR, puis deux fois par SINGLETON. DÉCORATEUR est intégré en identifiant COMPOSANTVISUEL avec COMPOSANT, VUETEXTE avec COMPOSANTCONCRET et DECORATEURBORD à la fois avec DÉCORATEURCONCRET et DÉCORATEUR. Les deux enrichissements par SINGLETON ont lieu par identification avec la même classe VUETEXTE en identifiant son attribut UNIQUEINSTANCE à l'attribut INSTANCE du DP.[26]
Dans le résultat de cet enrichissement exposé à la figure 3.20, l'enrichissement par DÉCORATEUR a ajouté l'opération SAMPLEOPERATION dans COMPOSANTVISUEL, DECORATEURBORD et VUETEXTE, mais il n'a pas propagé l'opération DESSINE de la classe COMPOSANTVISUEL vers ses spécialisations. De même, les enrichissements par SINGLETON ont ajouté OPÉRATION1 et OPÉRATION2 qui n'étaient pas désirées.

2. Plus proche de l'implémentation que les outils précédents, Borland Together considère qu'une classe et une interface sont des éléments fondamentalement différents[27] et sa fonctionnalité d'enrichissement interdit donc d'identifier une classe avec une interface.
   Ce type d'identification, par exemple une classe GRAPHIC avec l'interface COMPOSANT d'un COMPOSITE, ne provoque ni message d'erreur ni transformation de la nature de l'élément ; elle entraîne simplement l'ajout d'un nouvel élément, comme une interface portant un nouveau nom "Interface1"[28] dans l'exemple qui nous occupe.
3. La notion d'historique est absente alors même que les différentes transformations sont matérialisées dans le diagramme, mais sans ordre.
4. Les préconditions n'existent pas si ce n'est l'adéquation du type d'élément avec l'élément identifié. Les règles d'intégration sont pourtant suffisament rigides pour ne pas permettre de créer un diagramme syntaxiquement faux. Si l'enrichissement tente de créer un diagramme syntaxiquement incorrect, une erreur est générée. Par contre, la logique des DP, elle, n'est pas respectée. L'utilisateur peut facilement créer différentes aberrations : une classe plusieurs fois SINGLETON ou une classe jouant plusieurs rôles incompatibles dans un DP.
   L'exemple d'enrichissement illustré par les figures 3.18 et 3.20 montre, d'une part, la possibilité d'attribuer deux fois le rôle de SINGLETON à la classe VUETEXTE et, d'autre part, celui d'identifier la classe DECORATEURBORD à la fois au rôle de DÉCORATEUR CONCRET et à celui de DÉCORATEUR. La tentative de faire hériter DECORATEURBORD d'elle-même provoque une erreur illustrée en 3.19. Toutefois, le résultat final

opération (ou le type d'un attribut) dans la classe la plus générale, mais cette manipulation ne change pas automatiquement le type des membres hérités dans les classes plus spécialisées et crée des erreurs telles que des implémentations de méthodes qui ne retournent pas le même type que dans l'interface.

[26] Cet exemple est inspiré de celui présenté dans le livre *Design Patterns* [GoFDP, p. 205].

[27] Ce qui est vrai formellement : classe, interface et type de données (DataType) formant les trois "Classifier Elements", c'est-à-dire les trois éléments de base d'un diagramme de classes comme spécifié dans la norme UML ("3.21 Classifier", *in OMG Unified Modeling Language Specification* version 1.5 [UML, p. 3-35].)

[28] Le nom donné est "Interface" suivit d'un chiffre de façon à obtenir un nom différent de ceux déjà utilisés pour une classe ou une interface.

montre une classe DecorateurBord listée deux fois parmi les “participants” du “pattern” avec deux rôles différents et n’ayant pas hérité de ComposantVisuel. Ce diagramme marqué du pattern Décorateur n’en est donc pas un : il n’induira pas les conséquences attendues.

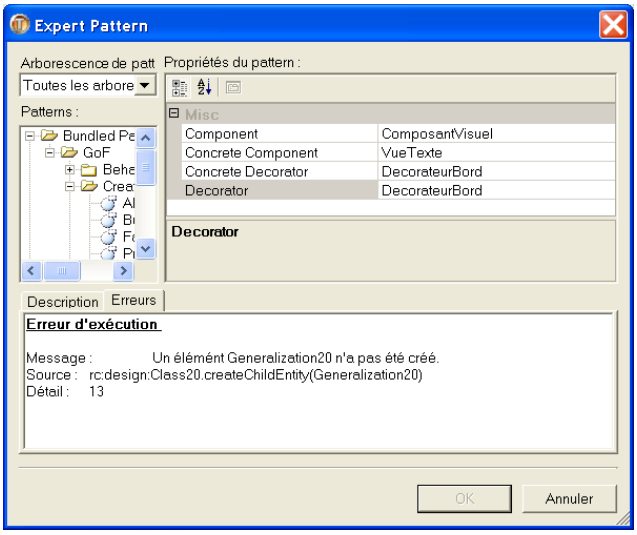

Fig. 3.19: Borland Together – inconvénients – erreur d’intégration

5. Enfin, un enrichissement par identification ne modifie jamais l’élément identifié. Si un utilisateur identifie une classe avec une classe stéréotypée du diagramme de structure, la classe résultante ne devient pas stéréotypée ; de même, un élément non statique ne devient pas statique, la visibilité reste inchangée, etc.
   Les figures 3.18 et 3.20 montrent l’enrichissement par Singleton de VueTexte en identifiant à l’attribut instance du diagramme de structure, l’attribut uniqueInstance non statique et de visibilité publique. Celui-ci n’est pas transformé et VueTexte dans le diagramme enrichi ne réalise pas l’idée du Singleton.

En définitive, Borland Together semble d’un premier abord présenter bien plus d’avantages que DPA Toolkit ou Enterprise Architect, notamment celui d’introduire, sous la forme d’un nouvel élément “pattern”, une référence claire aux enrichissements dans le diagramme. Néanmoins, les limites observées sont semblables à celles de ses prédécesseurs et tiennent, pour la plupart, au fait que l’enrichissement ajoute des éléments, mais ne les transforme pas. Après un enrichissement, l’utilisateur devra fréquemment corriger son diagramme pour obtenir un diagramme conforme au but de l’enrichissement.

Conclusion

Les trois outils CASE étudiés sous l’angle de l’enrichissement d’un diagramme de classes par DP montrent des comportements et des limites similaires.

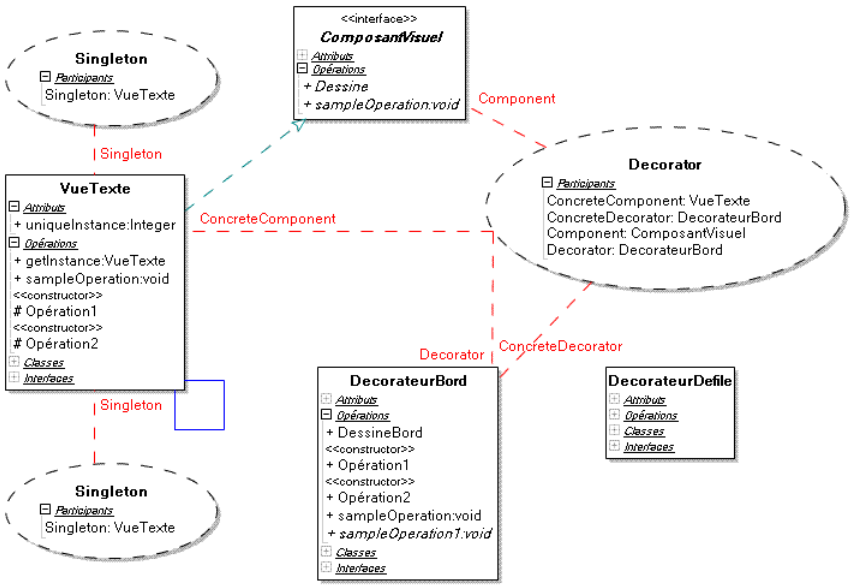

FIG. 3.20: Borland Together – inconvénients – diagramme enrichi

Ainsi, l'enrichissement a-t-il toujours lieu par ajout de nouveaux éléments ou par identification d'éléments du diagramme de base avec des éléments du diagramme de structure.

En outre, lors de l'enrichissement, chacun des logiciels, quoique de manière différente, accompagne la manipulation d'une méta-information afin de guider l'utilisateur dans ses choix, mais en marque aussi certains éléments avec le but de rappeler le rôle de l'élément dans le DP utilisé.

Au rang des limites, nous avons identifié quelques problèmes récurrents ; premièrement, des éléments sont ajoutés systématiquement sans choix de l'utilisateur quant à leur quantité ou leurs qualités (comme leur nom, leur visibilité ou leurs stéréotypes), ce qui dénote un manque de souplesse plus ou moins marqué selon l'outil étudié.

Ensuite, les préconditions sur les enrichissements sont assez faibles et permettent de créer des diagrammes qui ne respectent pas l'esprit du DP choisi quand la validité de la syntaxe elle-même n'est pas compromise.

De plus, l'historique des enrichissements est limité ou inexistant et ne permet jamais d'utilisation dérivée.

Ajoutons à ces trois limites que les éléments identifiés sont toujours laissés tels quels dans le diagramme sans changement de nature ni de qualité.

De surcroît, précisons que, en règle générale, la procédure d'enrichissement ne prend pas d'action spécifique à un DP donné [29].

[29] Par exemple, elle ne s'assure pas que l'attribut de rôle UNIQUEINSTANCE d'un SINGLETON doit être statique ou que les classes de rôle FEUILLE dans un COMPOSITE implémentent bien les opérations de leur interface COMPOSANT.

Finalement, l'enrichissement par quelque design pattern que ce soit fonctionne toujours sur les mêmes principes : ajout de certains éléments d'un diagramme de structure spécifique et non-ajout des éléments identifiés. Cette procédure permet de créer assez facilement un nouvel enrichissement dans la mesure où cela ne revient qu'à créer un diagramme spécifique agrémenté de quelques propriétés à respecter lors de l'enrichissement telles que, par exemple, le droit d'un élément d'être identifié ou non.

Pourtant, les diagrammes de structures sont des exemples d'utilisation d'un DP et pas des méta-modèles des DP. En toute généralité, ils ne peuvent pas être intégrés tels quels dans un diagramme.

Nous pensons que chaque enrichissement par un DP spécifique devrait faire l'objet d'un procédure propre à ce DP. Cette procédure pourrait ajouter des éléments ou transformer les éléments existant de manière spécifique selon le rôle de l'élément dans le DP.

Bien que le nombre d'outils étudiés soit limité, nous n'avons pas découvert à ce jour d'outil CASE qui applique ce principe. Par conséquent, nous considérons que les conclusions de cette étude justifient le développement d'une ébauche d'outil qui permette de critiquer cette idée à la lumière d'une implémentation.

Troisième partie

Proposition

Chapitre 4

Approche transformationnelle

L'état de l'art précédent (chap. 3) nous a amené à envisager une alternative à l'enrichissement d'un diagramme par simple ajout d'éléments. Plutôt que de considérer un élément identifié comme un élément déjà inséré – et donc non modifiable – nous souhaitons voir cet élément transformé par l'enrichissement.

Plus encore, puisque nous considérons que la transposition pure et simple du diagramme de structure d'un DP est insuffisante, nous souhaitons que chaque enrichissement par un DP particulier fasse l'objet d'une procédure particulière comprenant ajouts et transformations d'éléments. C'est ce que nous appelons l'approche transformationnelle.

Utiliser cette approche entraîne inéluctablement une perte en généricité car une procédure spécifique à chaque DP doit être développée. En contrepartie, elle devrait permettre une meilleure adéquation au cas particulier de chaque DP.

Afin d'implémenter cette approche, nous avons besoin de pouvoir manipuler le diagramme de classes sous une forme logique ; nos besoins peuvent se résumer de cette manière :

1. Nous devons pouvoir acquérir la connaissance du diagramme de base à enrichir.
2. Nous devons pouvoir définir des procédures d'enrichissement et donc :
   - nous devons devons pouvoir manipuler notre connaissance du diagramme de base.
3. Nous souhaitons poser quelques gardes-fous :
   - en vérifiant la validité syntaxique du diagramme de base avant et après manipulation,
   - en vérifiant des préconditions à un enrichissement.

La question de départ propose Prolog pour nous aider à combler ces besoins.

Prolog est le plus connu des langages répondant au paradigme de programmation logique[1]. Bien que nous tenions pour acquises la connaissance des bases

[1] La plupart des autres langages répondant à ce paradigme sont des descendants de Prolog.

de ce paradigme et la connaissance des concepts manipulés par Prolog tels que atome, arité, procédure, clause ou but[2], nous reviendrons sur certains d'entre eux afin de mettre en valeur les qualités qui justifient le choix de ce langage pour notre développement.

4.1 Programmation déclarative

La programmation procédurale implique pour le programmeur de dire à la machine ce qu'elle doit faire. Il doit écrire le "comment" en utilisant un algorithme.

Par ailleurs, la programmation déclarative, elle, implique de décrire les relations entre les entités manipulées. Le programmeur doit écrire le "quoi". La programmation est descriptive et permet donc de se concentrer sur le problème plutôt que sur la solution.

Un tel type de programmation consiste à représenter un système de connaissance, un *monde*. Dans ce *monde* devront être déclarés des faits et des règles.

La logique des prédicats du premier ordre permet une telle représentation à l'aide de clauses[3] auxquelles sont attribuées des valeurs de vérité : vrai ou faux. Ces clauses peuvent représenter des faits[4] ou des règles ; leur valeur de vérité est induite par le *monde* représenté : si l'interprétation de la clause dans le système de connaissance qu'on souhaite représenter est vrai, alors la clause prend la valeur de vérité vrai.

C'est sur cette logique qu'est basé Prolog, mais en utilisant une forme restreinte des clauses : les clauses de Horn[5], des clauses contenant au plus un atome non nié.

Soient α, β, γ des atomes, la formule sous forme non clausale :

$$(\beta \wedge \gamma) \Rightarrow \alpha$$

peut se réécrire sous forme d'une clause de Horn :

$$\alpha \vee \neg\beta \vee \neg\gamma$$

qui peut s'écrire :

$$\alpha \leftarrow \beta, \gamma$$

ce qui pourrait se lire, par exemple *si β et γ sont vrais alors α est vrai.*

En toute généralité, les clauses de Horn peuvent s'écrire :

$$\alpha \leftarrow \beta, \gamma, \delta \ldots$$

où $\alpha, \beta, \gamma, \delta \ldots$ sont des atomes, éventuellement absents.

[2]Ces notions et bien d'autres sont présentées dans le cours du Professeur J.–M. Jacquet, *Théorie des Langages, paradigmes de programmation (2e partie)* [PdP2].

[3]La forme clausale est une façon d'écrire des formules bien formées, c'est-à-dire de combiner des atomes à l'aide de connecteurs logiques.

[4]Les faits sont représentés par des clauses fermées, c'est-à-dire des clauses composées d'atomes clos, des atomes qui ne contiennent pas de variables.

[5]Les clauses de Horn sont aussi appelées "definite clauses".

Un programme Prolog est un ensemble de clauses de Horn de valeur de vérité vrai. Le programmeur dispose donc d'un outil pour décrire un *monde* à partir de faits et de règles avérés.

La syntaxe Prolog est proche de la syntaxe des clauses utilisée ci-dessus. Les $\leftarrow$ y sont rempacées par " :-" et la terminaison des clauses est indiquée par un ".".

A titre d'exemple, considérons les faits suivants d'un *monde* "diagramme de classes"[6] dont nous souhaitons acquérir le système de connaissances en Prolog : le diagramme est composé d'une classe nommée "myClass" possédant un attribut nommé "myAttribute". Nous pouvons décomposer ce système de connaissances de cette manière :

- Une classe existe (désignons la "c").
- Le nom de cette classe est "myClass".
- Un attribut existe (désignons le "a").
- Cet attribut a pour nom "myAttribute".
- Cet attribut appartient à la classe précédemment définie.

Sous forme de clauses de Horn, ces faits pourraient se traduire[7] :

$$class(c) \leftarrow \tag{4.1}$$
$$class\_name(c, myClass) \leftarrow \tag{4.2}$$
$$attribute(a) \leftarrow \tag{4.3}$$
$$member\_name(a, myAttribute) \leftarrow \tag{4.4}$$
$$has(c, a) \leftarrow \tag{4.5}$$

Le programme Prolog correspondant à ces clauses s'écrit :

```
class(c).
class_name(c,myClass).
attribute(a).
member_name(a,myAttribute).
has(c,a).
```

Mais encore, ce système de connaissances peut s'accompagner de règles ; considérons la règle suivante : un membre est un attribut ou une opération. Cette règle signifie bien "tout membre est soit un attribut, soit une opération" et peut être décomposée ainsi :

- Un attribut est un membre.
- Une opération est un membre.

[6] Afin de lever toute ambiguité, précisons que le *monde* "diagramme de classes" en question est un humble diagramme de classes particulier et non un méta-modèle des diagrammes de classes.

[7] Les chiffres entre parenthèses à droite de chaque clause n'en font pas partie, ils ne sont que des références aux formules utilisées afin de faciliter la lecture.

Ces règles peuvent être traduites sous forme de clauses de Horn :

$$member(X) \leftarrow attribute(X) \tag{4.6}$$
$$member(X) \leftarrow operation(X) \tag{4.7}$$

Nous pouvons alors ajouter à notre programme précédent les clauses suivantes :

```
member(X) :-
              attribute(X).
member(X) :-
              operation(X).
```

Ce programme à présent composé de faits et de règles constitue une représentation des connaissances du *monde* "diagramme de classes". Précisons que Prolog considère que le programme représente toutes les connaissances de ce *monde* et que tout ce qui ne peut en être déduit est faux, c'est-à-dire que Prolog utilise l'hypothèse de monde clos (*closed world assumption* ou *cwa*), définie comme suit :

> "L'hypothèse de monde clos est un mécanisme qui nous permet de tirer des conclusions négatives basées sur le manque d'information positive. Le *cwa* est une règle qui est utilisée pour dériver l'expression $\neg A$ à condition que A soit une formule atomique close qui ne puisse être dérivée par les règles d'inférence sur le système utilisé [...]."[LPP, p. 60]

Donc, Prolog permet bien d'acquérir la représentation d'un système de connaissances, mais, plus encore, il permet de tirer les conséquences du programme en vérifiant la véracité d'une nouvelle assertion. En effet, une clause particulière appelée un but peut être posée. Un but est une clause dont l'atome de tête est absent – une façon de poser une interrogation de manière déclarative. En effet, le but est vrai si chacun de ses atomes est vrai et donc la formule suivante :

$$\leftarrow \alpha, \beta$$

peut être lue *Est-ce que* α *et* β *sont vrais ?*

4.2 Unification

L'unification est le processus qui permet de questionner notre représentation des connaissances, c'est-à-dire de résoudre un but fixé.

4.2.1 Substitution

Une substitution est un ensemble qui décrit des remplacements de variables par des termes. Elle se définit comme suit :

Soient n$\in \mathbb{N}$, t_i des termes et X_i des variables, une substitution est un ensemble $\{X_1/t_1, \ldots, X_n/t_n\}$ tel que

$X_i \neq X_j \, \forall i, j \in \mathbb{N}$ tel que $1 \leq i, j \leq n$ et $i \neq j$ (sans ambiguité) et
$X_i \neq t_i \forall i \in \mathbb{N}$ (sans rempacement trivial).

Par exemple, $\{X/a\}$ ou $\{X/has(c, a)\}$ sont des substitutions.

L'application X/θ d'une substitution θ à une variable X est définie comme suit :

$$X\theta := \{ \begin{array}{ll} t & \text{si } X/t \in \theta \\ X & \text{sinon} \end{array}$$

ce qui nous permet de définir l'application $t\theta$ d'une substitution θ à un e-terme[8] t par l'e-terme obtenu en appliquant θ à chaque variable de t.

Voici un exemple d'application : $member(X)\{X/has(c, a)\} = member(has(c, a))$.

La combinaison de deux substitutions existe également :

Soient α et β des substitutions telles que :

$$\alpha = \{X_1/t_1, \ldots, X_n/t_n\}$$
$$\beta = \{Y_1/u_1, \ldots, Y_n/u_n\}$$

la composition $\alpha\beta$ de α et β est obtenue à partir de

$$\{X_1/t_1\beta, \ldots, X_n/t_n\beta, Y_1/u_1, \ldots, Y_n/u_n\}$$

en y supprimant tout $X_i/t_i\beta$ tel que $X_i = t_i\beta$ et tout Y_j/u_j tel que $Y_j \in \{X_1, \ldots, X_n\}$

4.2.2 MGU

C'est à partir de la notion de substitution qu'on définit la notion fondamentale d'unification, le mécanisme d'inférence permettant de rendre deux e-termes syntaxiquement équivalents.

Soient deux e-termes t et u, ils s'unifient si et seulement s'il existe une substitution σ telle que $t\sigma = u\sigma$; σ est alors appelé *unificateur* de t et u.

Un unificateur σ est dit l'unificateur le plus général de deux e-termes ou "most general unifier" ou encore mgu, si et seulement si σ est plus général que tout autre unificateur de ces e-termes, c'est-à-dire si et seulement si $\forall$ unificateur $\alpha \neq \sigma, \exists$ unificateurβ tel que $\beta = \sigma\alpha$. Notons que les mgu sont uniques *modulo* renomination des variables.

De plus, nous pouvons affirmer grâce à l'algorithme d'unification de Herbrand[9] que deux e-termes sont unifiables si et seulement s'ils possèdent un mgu que cet algorithme permet de trouver.

[8] Un e-terme, ou terme étendu, est un terme ou un atome.

[9] L'algorithme d'unification de Herbrand est présenté au point 2.4 du cours *Théorie des Langages, paradigmes de programmation (2e partie)* [PdP2].

A titre d'exemple, considérons les termes $has(c, a)$ et $has(c, b)$: ils ne sont pas unifiables. Par contre, on peut unifier les termes $has(X, a)$ et $has(c, Y)$ avec, par exemple, l'unificateur $\{X/c, Y/a, Z/b\}$, mais ce n'est pas le mgu qui est $\{X/c, Y/a\}$.

4.2.3 SLD-dérivation

Grâce à l'unification, nous sommes capable de répondre à la question posée par un but. En effet, en inférant ce but avec les clauses connues pour vraies, nous pouvons arriver à conclure qu'un but est vrai ou non[10]. Ces inférences sont réalisées par le mécanisme d'unification sous une procédure connue comme la SLD-dérivation.

Soient i, k, n et m $\in \mathbb{N}$ avec $1 \leq i \leq m$, $A_1, \ldots A_m$, $B_1, \ldots, B_n$ et H des atomes et A_i s'unifiant à H avec le mgu σ, alors :

$$\begin{gathered} \leftarrow A_1, \ldots, A_{i-1}, A_i, A_{i+1}, \ldots, A_m \\ H \leftarrow B_1, \ldots, B_n \\ \downarrow \\ \leftarrow (A_1, \ldots, A_{i-1}, B_1, \ldots, B_n, A_{i+1}, \ldots, A_m)\theta \end{gathered}$$

est une SLD-résolution[11] ; en appliquant successivement cet exercice, ce qu'on appelle réaliser une SLD-dérivation, jusqu'à obtenir la clause vide[12] $\square$, la preuve est faite du but[13]. De plus, la conbinaison des mgu produits $\sigma_1\sigma_2 \ldots \sigma_k$ forme une solution appelée Computed Answer Substitution ou CAS.

A titre d'exemple, considérons la question "La classe de nom myClass a-t-elle un membre (et quel est le nom de ce membre) ?" dans le contexte du programme défini en 4.1. Nous pouvons traduire cette question par le but suivant :

$$\leftarrow class\_name(C, myClass), member(X), has(C, X), member\_name(X, N) \tag{4.8}$$

et le dériver comme suit[14] :

$$\begin{gathered} \leftarrow class\_name(C, myClass), member(X), has(C, X), member\_name(X, N) \\ \downarrow 4.1 \end{gathered}$$

[10] En toute généralité, certains cas de SLD-dérivation ne permettent pas d'arriver à une conclusion.

[11] L'acronyme signifie *Linear resolution for Definite clauses with Selection function.*

[12] Si au terme d'une SLD-dérivation, la clause obtenue est non-vide et ne peut être résolue par aucune autre clause du sytème, la dérivation a échoué.

[13] Pour être précis, la SLD-dérivation est une preuve par l'absurde puisque le but, sous forme déclarative, peut s'écrire
$\forall X_i$, i $\in \mathbb{N}$, tels que X_i sont des variables des atomes $\alpha, \beta, \gamma \ldots$, on a $(\neg\alpha \vee \neg\beta \vee \neg\gamma \ldots)$
et que la SLD-dérivation le conduit à $\square$, la conjonction vide d'atome qui est fausse (par cwa) et qui donc prouve la négation du but, c'est-à-dire
$\exists X_i$, i $\in \mathbb{N}$, tels que X_i sont des variables des atomes $\alpha, \beta, \gamma \ldots$ et $(\alpha \wedge \beta \wedge \gamma \ldots)$.

[14] Précisons pour la compréhension de cette dérivation que les variables n'ont pour portée que la clause où elles sont utilisées. Donc, le "X" de la clause 4.6 et celui du but 4.8 ne sont pas les mêmes et lors de la résolution, nous renommons la variable de l'équation 4.6 par X_1 afin d'éviter toute ambiguité.

$$\leftarrow (member(X), has(C, X), member\_name(X, N))\,\{C/c\}$$
$$\Leftrightarrow$$
$$\leftarrow member(X), has(c, X), member\_name(X, N)$$
$$\downarrow 4.6$$
$$\leftarrow (attribute(X_1), has(c, X), member\_name(X, N))\,\{X/X_1\}$$
$$\Leftrightarrow$$
$$\leftarrow attribute(X_1), has(c, X_1), member\_name(X_1, N)$$
$$\downarrow 4.3$$
$$\leftarrow (has(c, X_1), member\_name(X_1, N))\,\{X_1/a\}$$
$$\Leftrightarrow$$
$$\leftarrow has(c, a), member\_name(a, N)$$
$$\downarrow 4.5$$
$$\leftarrow (member\_name(a, N))\,\{\}$$
$$\downarrow 4.4$$
$$\leftarrow ()\,\{N/myAttribute\}$$
$$\Leftrightarrow$$
$$\square$$

et obtenir non seulement la preuve du but, mais aussi la solution,

$$\text{CAS} = \{C/c\}\{X/X_1\}\{X_1/a\}\{\}\{N/myAttribute\}$$

qui peut se comprendre ainsi : le but exprimé par l'équation 4.8 est résolu avec $C = c$, $X = a$ et $N = myAttribute$. La question posée a donc pour réponse : "la classe de nom myClass a un membre de nom myAttribute".

Ajoutons que la SLD-dérivation n'est pas forcément déterministe ; si des alternatives existent [15], la dérivation peut échouer alors qu'une solution existe ou encore réussir en ignorant les autres solutions. Contrer ce biais implique de revenir en arrière pour réessayer les alternatives ; cette méthode s'appelle le *backtracking*.

4.2.4 Avantages

La SLD-résolution possède des qualités importantes qui sont aussi les qualités de son implémentation dans Prolog à savoir cohérence, complétude et semi-décidabilité.

Cohérence

> "La cohérence (soundness) est une propriété essentielle qui garantit que les conlusions produites par le système sont correctes. La correction dans ce contexte signifie qu'elles sont les conséquences

[15] Comme le choix de la clause 4.6 dans notre exemple qui aurait bien pu être celui de la clause 4.7, puisqu'elle aussi unifie le terme recherché.

> logiques du programme. C'est-à-dire qu'elles sont vraies dans tous les modèles du programme."[LPP, p. 49]

Un système de clauses de Horn, tel qu'un programme Prolog, est cohérent via la SLD-dérivation car les CAS sont des conséquences logiques du système. Donc, les conclusions tirées du programme Prolog peuvent être considérées comme valides dans le *monde* représenté par le programme. Plus généralement, elles sont valides dans tous les *mondes* desquels le programme est une représentation.

Complétude

Un programme Prolog est complet sous la SLD-résolution, ce qui signifie que toutes les réponses correctes d'un but donné peuvent être obtenues par SLD-résolution. Cette complétude est rendue possible par le *backtracking*[16].

Semi-décidabilité

Hélas, la SLD-dérivation peut ne pas se terminer dans certains cas particuliers, lorsque le but n'est pas déduisible du système de clause.

Par exemple, si le système de clauses contient une tautologie telle que

$$member(X) \leftarrow member(X)$$

et que le but à résoudre utilise cette clause, comme

$$\leftarrow member(a)$$

alors, la SLD-dérivation peut boucler infiniment et il est impossible de déterminer si $member(a)$ est vrai ou faux.

4.3 Méta-programmation

Prolog peut être utilisé pour décrire un autre programme et en particulier un programme Prolog ; c'est à ce titre que nous le qualifions de méta-langage.

De même, nous qualifions de "méta-programme", un programme capable de manipuler un autre programme, par exemple lui-même.

Des interpréteurs de Prolog[17] proposent d'intéressants prédicats pré-intégrés afin de manipuler le programme interprété[18]. Grâce à ceux-ci, un programme

[16] Prolog effectue une recherche en profondeur d'abord avec *backtracking* afin de n'omettre aucune solution possible.

[17] Dont celui que nous avons choisi pour notre développement : *SWI-Prolog* version 5.4.7 par Jan Wielemaker, Copyright (c) 1990-2003 University of Amstredam. `http://www.swi-prolog.org`

[18] De tels interpréteurs sont appelés des interpréteurs méta-circulaires.

Prolog peut notamment s'ajouter ou se supprimer des clauses. Voici deux prédicats[19] proposés à cet effet :

assert/1 est utilisé pour modifier le programme pendant l'éxécution d'un but ; ASSERT/1 permet d'ajouter dynamiquement une nouvelle clause au programme.
"assert(+*Term*) déclare un fait ou clause dans la base de données[20]. *Term* est déclaré comme le dernier fait ou clause du prédicat correspondant."[SWI, p. 86]

retract/1 est utilisé pour enlever dynamiquement une clause du programme pendant l'éxécution d'un but.
"retract(+*Term*) quand *Term* est un atome ou un terme il est unifié avec le premier fait ou clause unifiante dans la base de données. Le fait ou clause est enlevé de la base de données."[SWI, p. 86]

Grâce à ces capacités de méta-langage peut être construit un méta-programme Prolog capable de changer la représentation du système de connaissances qu'il porte.

Conclusion

Prolog nous offre donc tous les atouts pour combler les besoins exprimés au début de ce chapitre :

1. Nous pouvons acquérir la connaissance du diagramme de base à enrichir sous forme de faits grâce au concept de programmation déclarative.
2. Nous pouvons manipuler notre connaissance du diagramme de base grâce à la méta-programmation qui permet de modifier dynamiquement les faits connus et donc :
   – nous pouvons définir des procédures d'enrichissement.
3. Nous pouvons vérifer la validité syntaxique du diagramme de base avant et après manipulation ou encore les préconditions des enrichissements en déclarant des règles de validités et de préconditions vérifiées grâce au processus d'unification :
   – nous pouvons donc poser les gardes-fous souhaités.

[19] Les prédicats sont définis par leur nom et leur arité ; écrire ce nom suivit de "/" et de l'arité est donc couramment utilisé pour les représenter. Une autre façon de les représenter existe ; afin de décrire plus précisément un prédicat, ses arguments sont représentés par une variable dont le nom tente d'être significatif. La variable est alors précédée d'un symbole parmi "-", "+" ou " ?" qui signifient repectivement que le paramètre est susceptible d'être un *output* (la valeur de la variable est mise à jour après passage par le prédicat), un *input* (la variable doit être valuée avant passage par le prédicat) ou les deux invariablement.

[20] La base de données de l'interpréteur Prolog qui contient les clauses formant le programme. (C'est nous qui notons.)

Chapitre 5

Modèle de transformation

Dans le cadre de la programmation déclarative de Prolog, une bonne pratique est, pensons-nous, de modéliser le système de connaissances à acquérir et à manipuler. A cette fin, nous proposons d'abord un méta-modèle simplifié des diagrammes de classes et ensuite un modèle des transformations.

Dans un second temps, nous utilisons les concepts découverts par cette modélisation pour décrire l'implémentation de notre système de transformation de diagrammes qui découle de cette même modélisation.

Nous traitons de cette implémentation à travers la description de sa conception physique pour ensuite la valoriser par une interface homme-machine visuelle ; c'est DB-Main qui jouera ce rôle. Nous décrivons donc, pour clore ce chapitre, l'intégration de notre système de transformation et de DB-Main.

5.1 Méta-modèle des diagrammes de classes

Notre modélisation est réalisée à l'aide d'un modèle basé sur trois concepts principaux : l'entité, l'association et l'attribut. Il s'agit du modèle entité-association, aussi appelé Entité-Relation-Attribut, d'où l'acronyme ERA. Ce modèle permet de décrire un domaine de données – habituellement en vue de l'intégration de ces données dans une base de données. Nous supposons ces concepts connus et renvoyons pour plus d'informations au cours d'*Ingénierie des Bases de données*[1].

Ces précisions établies, examinons la figure 5.1 ; elle montre un méta-modèle[2] simplifié des diagrammes de classes. Les entités de ce méta-modèle correspondent pour la plupart aux éléments principaux des diagrammes de classes : classe, associations, généralisations et membres. Toutefois, nous avons ajouté une super-entité "Transformable" afin de désigner les éléments que les transformations doivent être capables de modifier.

[1] Jean-Luc Hainaut, *Ingénierie des Bases de données*, "Partie 5. Le modèle entité-association"[BD]

[2] Un méta-modèle est entendu comme un modèle de modèle ; en l'occurence, comme un diagramme de classes est un modèle, son modèle est un méta-modèle.

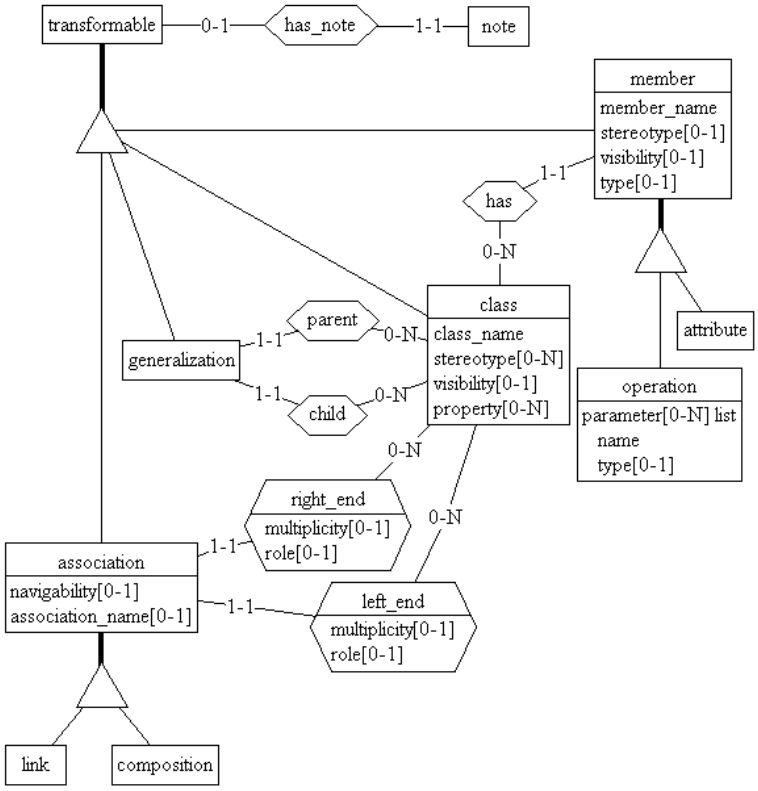

FIG. 5.1: Méta-modèle simplifié des diagrammes de classes

Notons que la portée de ce méta-modèle est limitée à certains diagrammes de classes car des notions y sont omises afin de préserver la simplicité du propos. Ainsi, les concepts d'interface[3], de package, d'instance, de dépendance, d'association n-aire, d'association de classes (*class association*) et d'autres encore ne peuvent être modélisées par ce moyen[4].

Voici la description des concepts représentés dans le schéma de la figure 5.1 :

Transformable : TRANSFORMABLE[5] est le super-type des éléments transformables.
Tout élément de ce type peut avoir une NOTE via l'association HAS_NOTE.

Note : une NOTE est un texte libre associé à un TRANSFORMABLE.

Member : un MEMBER représente un membre, élément transformable. Le nom du membre est représenté par l'attribut MEMBER_NAME. Un membre peut avoir un stéréotype, une visibilité et/ou un type ; ces concepts sont

[3]La notion d'interface sera recouverte par celle d'une classe stéréotypée INTERFACE.

[4]Un méta-modèle plus complet peut être trouvé *in OMG Unified Modeling Langage Specification*, "5. UML Model Interchange"[UML]

[5]TRANSFORMABLE est aussi présent dans le modèle des transformations.

modélisés par les attributs STEREOTYPE, VISIBILITY[6] et TYPE[7].
Un membre appartient à une classe par l'association HAS.

Attribute : un ATTRIBUTE est un MEMBER ; il représente un attribut.

Operation : l'entité OPERATION est un MEMBER : il représente une opération. Une opération peut avoir aucun, un ou plusieurs paramètre(s) ayant chacun un nom et éventuellement un type : ces paramètres sont représentés par une liste PARAMETER composée des attributs NAME et TYPE.

Class : CLASS représente une classe, élément transformable. CLASS possède les attributs CLASS_NAME, STEREOTYPE, VISIBILITY et PROPERTY représentant respectivement le nom de la classe et ses éventuels stéréotypes, visibilité et propriétés.
Une classe peut avoir aucun, un ou plusieurs membre(s), ce qui est représenté par l'association HAS avec MEMBER. Une classe peut généraliser ou spécialiser aucune, une ou plusieurs autre(s) classe(s), ce qui est représenté, respectivement, par les associations PARENT et CHILD de CLASS avec GENERALISATION. Enfin, des classes peuvent être reliées par des associations (concept UML), ce qui est représenté par les associations (concept ERA) RIGHT_END et LEFT_END[8] avec l'entité ASSOCIATION. RIGHT_END et LEFT_END peuvent avoir une MULTIPLICITY pour la multiplicité et/ou un ROLE pour le nom de rôle[9].

Generalization : GENERALIZATION représente une généralisation, élément transformable.
Une généralisation relie toujours une classe générale à sa spécialisation comme représenté par PARENT et CHILD.

Association : une association entre deux classes est représentée par l'entité ASSOCIATION. Une association est un élément transformable qui possède éventuellement une navigabilité et/ou un nom, ce qui est représenté par les attributs NAVIGABILITY et ASSOCIATION_NAME
Les fins d'associations (concept UML) sont représentées par les associations (concept ERA) RIGHT_END et LEFT_END avec class.

Link : LINK représente une association de type lien (*link*).

Composition : COMPOSITION représente une association de composition. Dans ce cas, c'est l'association RIGHT_END qui détermine la classe composante[10]. Notons que nous ne définissons pas d'agrégation.

5.2 Modélisation des transformations

Après la modélisation des diagrammes de classes, considérons la modélisation des transformations de ces diagrammes.

---

[6] VISIBILITY est un texte restreint à "public", "private", "protected" ou "package".

[7] Un TYPE est un texte qui est soit le nom d'une classe du diagramme, soit un des noms suivant : "int", "char", "bool", "real" ou "string".

[8] RIGHT_END et LEFT_END représentent le concept UML de fin d'association (*association end*).

[9] Le nom de la fin d'association.

[10] C'est-à-dire la classe qui est composée de l'autre classe

Ces transformations, une par DP, sont décrites par un certain nombres de tâches qu'elles accomplissent sur le diagramme. De plus, à chaque transformation sont associés des rôles : des mots-clefs identifiant le rôle joué par tel ou tel élément au sein d'un design pattern. Enfin, chaque transformation a une précondition[11] qui contient les règles à vérifier sur le diagramme avant de tenter la transformation.

5.2.1 Historique

Nous souhaitons, dans un premier temps, conceptualiser une fonctionnalité d'historique, c'est-à-dire que le modèle comprenne une représentation des transformations effectivement appliquées au diagramme de classes.

A cette fin, nous introduisons le concept de *played_ transformation* : une transformation ayant été "jouée", c'est-à-dire ayant transformé le diagramme de classes. Après une transformation, une relation ternaire associe les rôles de cette transformation avec la *played_ transformation* créée et les transformables altérés par la transformation. Cette relation appelée *transformed* fournit le lien entre l'historique des transformations et les effets de celle-ci sur les transformables.

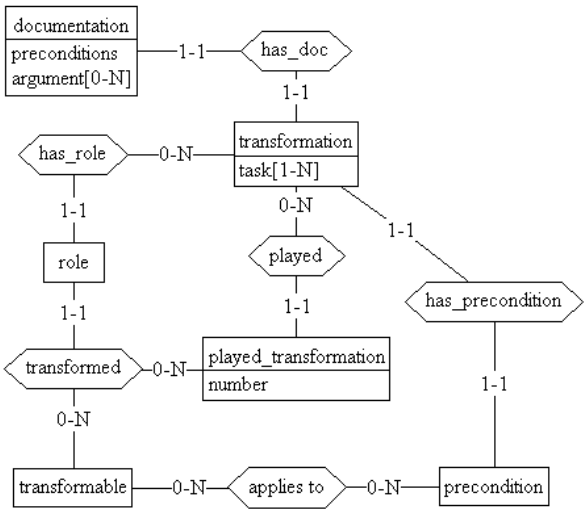

FIG. 5.2: Modèle des transformations – historique

Considérons la figure 5.2 qui représente le modèle des transformations avec sa partie gestion de l'historique. Le lien est fait avec le schéma de la figure 5.1 par l'entité TRANSFORMABLE. En effet, les transformations altèrent le diagramme de classes, c'est-à-dire qu'elles modifient, créent ou suppriment des transformables.

Voici la description des concepts de ce modèle des transformations et de gestion de l'historique :

[11] Nous désignons au singulier un ensemble de règles qui peut éventuellement être vide, mais que chaque transformation doit posséder.

Transformation : l'entité TRANSFORMATION représente une transformation. Une transformation possède des tâches représentées par l'attribut TASK ; cet attribut représente des tâches complexes et *ad hoc* à chaque transformation.
La relation HAS_DOC permet le lien avec l'entité DOCUMENTATION afin de maintenir une description de la transformation. Les rôles possibles d'une transformation sont accessibles par la relation HAS_ROLE avec l'entité ROLE. Les éventuelles *played_transformations* d'une transformation y sont reliées par l'association PLAYED. La précondition d'une transformation est accessible par l'association HAS_PRECONDITION.

Précondition : l'entité PRECONDITION représente la précondition d'une transformation associée à celle-ci par HAS_PRECONDITION.
Une précondition s'applique à un ensemble de transformables, ce qui est représenté par l'association APPLIES TO.

Documentation : chaque transformation est qualifiée d'une documentation[12] décrivant les préconditions et les éventuels arguments, ce qui est représenté par l'entité DOCUMENTATION et ses attributs PRECONDITIONS et ARGUMENT.
L'association HAS_DOC relie DOCUMENTATION à sa TRANSFORMATION.

Played_Transformation : l'entité PLAYED_TRANSFORMATION représente le concept du même nom. Les *played_transformations* sont numérotées dans l'ordre dans lequel les transformations ont eu lieu ; cette numérotation est symbolisée par l'attribut NUMBER.
Une PLAYED_TRANSFORMATION est toujours celle d'une transformation particulière ; le lien entre les entités est effectué par la relation PLAYED. Les *played_transformations* sont liées aux rôles de leurs transformations et aux transformables altérés par celles-ci via la relation TRANSFORMED qui attribue à un transformable transformé le rôle qu'il joue dans le DP en question.

Role : les rôles d'une transformation sont représentés par l'entité ROLE. Un rôle est relié à sa transformation par HAS_ROLE et aux transformables qui jouent effectivement ce rôle (ainsi qu'à la *played_transformation*) par TRANSFORMED.

Transformable : TRANSFORMABLE est l'entité décrite au point 5.1.
Un transformable joue un rôle pour chaque transformation par laquelle il a été affecté ; c'est l'association TRANSFORMED qui permet de le joindre dans le schéma. De plus, un transformable peut être utilisé dans la vérification de la précondition d'une transformation, ce qui est symbolisé par APPLIES TO.

5.2.2 Modèle de transformations

Nous avons, dans un premier temps, modélisé les transformations et leur historique. Dans un second temps, nous nous attachons à conceptualiser une fonctionnalité de publication des informations concernant ces transformations.

[12] Son but premier est une documentation interne à l'usage du programmeur.

En effet, notre représentation logique du diagramme n'est pas très pratique pour un utilisateur souhaitant manipuler des diagrammes sous forme visuelle, c'est pourquoi nous souhaitons pouvoir proposer une interface qui permet à une application externe – comme un logiciel de visualisation de diagrammes de classes – l'échange d'informations avec notre sytème de connaissances. Notamment, un utilisateur (homme ou machine) devrait pouvoir prendre connaissance de la façon d'appeler une transformation sans la connaître au préalable[13].

En conséquence, le modèle propose une description de la structure des transformations afin de pouvoir la publier. A chaque transformation, il associe une *signature* qui représente la manière de faire appel à la transformation. Cette *signature* est qualifiée d'un nom et possède aucun, un ou plusieurs paramètre(s), éventuellement facultatifs. Ces paramètres sont nommés et ils ont un ou plusieurs[14] type(s) parmi un nom d'opération, un nom d'attribut, un nom de classe, un littéral ou une liste de paramètres[15].

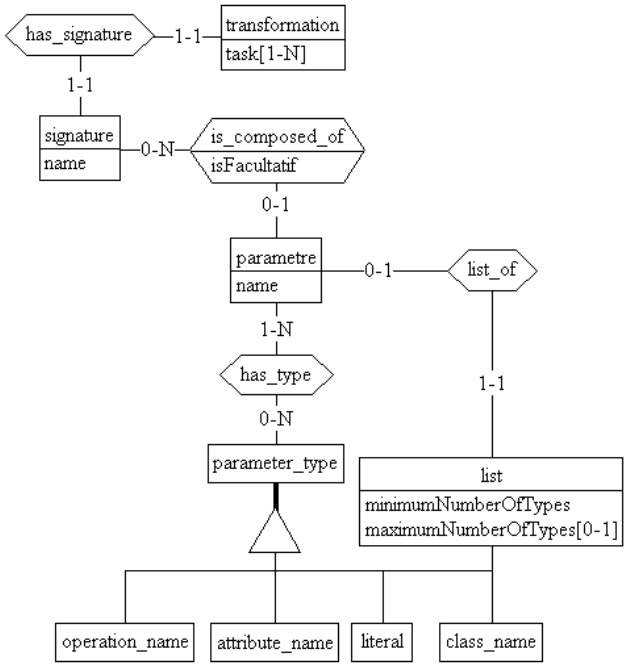

FIG. 5.3: Modèle des transformations – signature

La figure 5.3 représente un modèle centré sur la signature des transformations ; en voici la description des concepts :

Transformation : TRANSFORMATION est l'entité décrite précédemment en 5.2.1.

[13]Cette information pourrait être accompagnée d'une méta-information à propos des transformations et de leur DP, obtenue par la documentation décrite en 5.2.1.

[14]Cas rare où un paramètre doit associer plusieurs éléments

[15]Plus précisément, une liste d'éléments d'un type défini par un paramètre.

Une transformation a une signature ; le lien entre les entités est assuré par l'association HAS_SIGNATURE.

Signature : l'entité SIGNATURE représente une signature. Elle arbore un attribut NAME représentant son nom (le nom de la transformation).
Elle est associée à l'entité PARAMETRE via la relation IS_COMPOSED_OF qui possède un attribut ISFACULTATIF représentant le caractère facultatif ou non du paramètre dans la signature.

Parametre : un paramètre est représenté par l'entité homonyme. Son nom est représenté par l'attribut NAME.
Un paramètre fait partie soit d'une liste de paramètres, soit d'une signature ; ces relations sont représentées par IS_COMPOSED_OF et LIST_OF.
De plus, l'entité PARAMETRE est associée à l'entité PARAMETER_TYPE via la relation HAS_TYPE.

Parameter_type : PARAMETER_TYPE représente un type de paramètre. Les types possibles sont représentés par les entités OPERATION_NAME, ATTRIBUTE_NAME, CLASS_NAME, LITERAL et LIST représentant respectivement un nom d'opération, d'attribut ou de classe, un littéral et une liste de paramètres.

List : le type liste de paramètres est représenté par l'entité LIST qui possède un attribut MINIMUMNUMBEROFTYPES et un attribut facultatif MAXIMUMNUMBEROFTYPES. Ces attributs représentent les nombres minimum et maximum d'éléments que la liste peut recevoir lors de l'appel de la transformation.
Le paramètre des éléments de la liste est connu par l'association LIST_OF.

5.3 Conception physique

Les modèles découverts précédemment permettent d'envisager la mise au point de la conception physique d'un système de transformation de diagrammes de classes. Nous entendons par là une description plus proche de l'implémentation comprenant une représentation complète du système de connaissances acquis par le programme Prolog et une découpe de cette implémentation en une structure commode.

5.3.1 Modèle du système de connaissances

Le système de connaissances représenté dans la programmation déclarative se base sur les modèles précédemment décrits (voir le point 5.2). L'assemblage de ces modèles ainsi que quelques simplifications amènent au schéma illustré par la figure 5.4. Cet assemblage est évident en considérant les points de jonction du méta-modèle des diagrammes de classes et des modèles des transformations (historique et signature) que sont les entités TRANSFORMABLE et TRANSFORMATION.

La simplification la plus importante est celle qui range – abusivement, mais pratiquement – la généralisation parmi les associations. PARENT et CHILD sont identifiés respectivement à LEFT_END et RIGHT_END tandis que les attributs

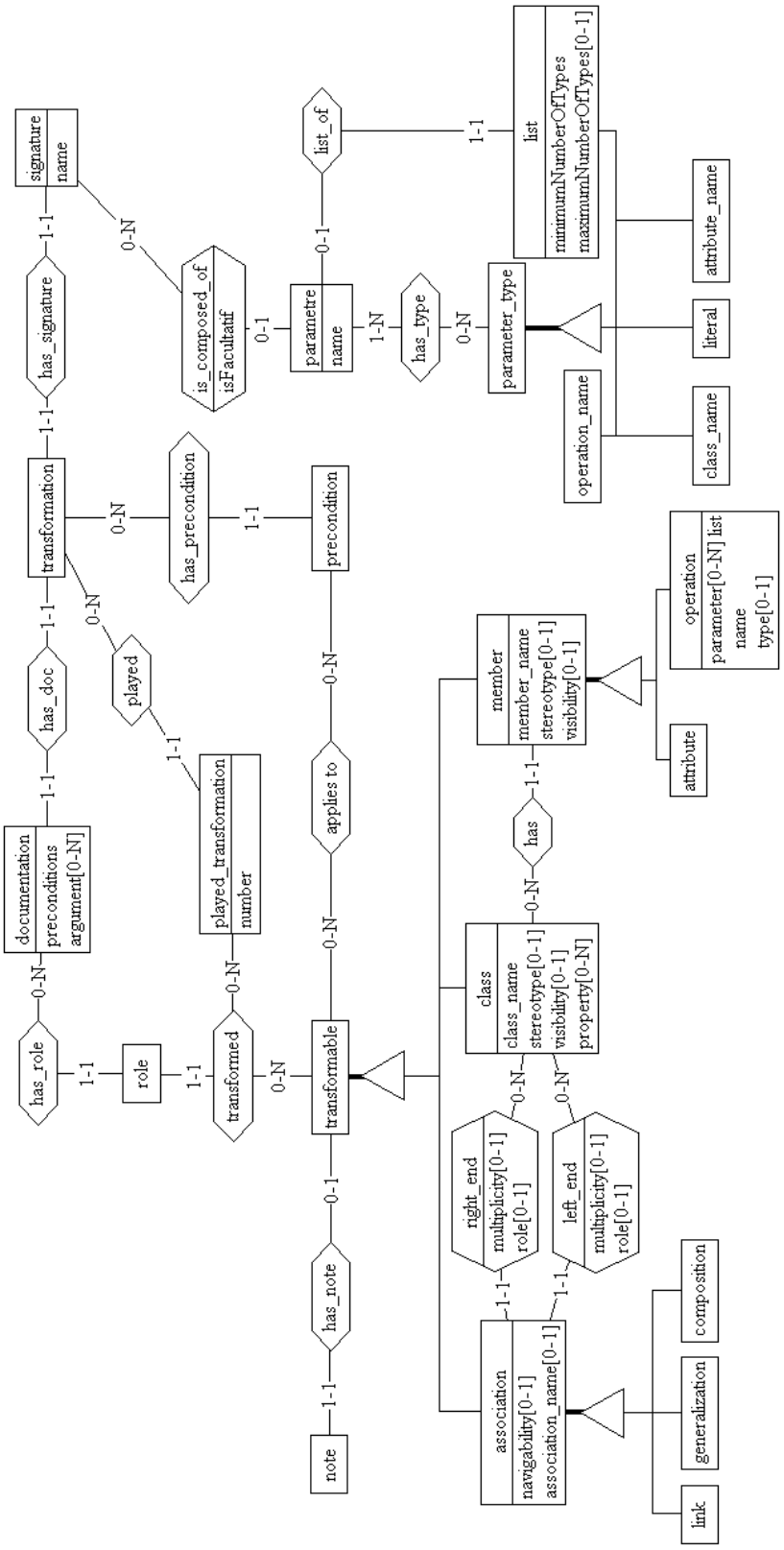

FIG. 5.4: Modèle complet du système de connaissances

inutiles à une généralisation deviennent facultatifs afin de ne pas être utilisés dans ce cas. Nous avons aussi omis les types des attributs et les types de retour des opérations .

5.3.2 Architecture

L'architecture de l'implémentation du système de connaissances modélisé est représenté à la figure 5.5 par un diagramme des composants[16]. Ce diagramme représente le *noyau* du système développé que nous définissons comme la partie centrale implémentant les fonctionnalités décrites ci-dessus et excluant l'interaction avec l'utilisateur (notamment la visualisation des diagrammes).

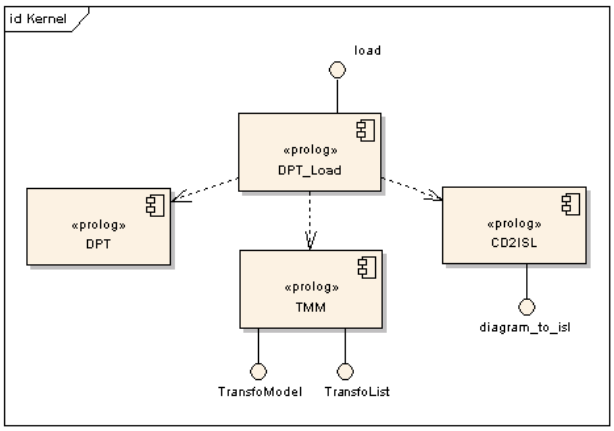

FIG. 5.5: Diagramme des composants – noyau

Cette architecture du *noyau* comprend quatre composants : DPT pour Design Pattern Transformer, DPT_Load pour Design Pattern Transformer Loader, CD2ISL pour Class Diagram to ISL file[17] et TMM pour Transformation Méta-Model.

Voici la description et le rôle de chacun de ces composants :

1. DPT_Load prend en charge les chargements des divers programmes Prolog. C'est un composant essentiellement technique dont la vocation première est de permettre à un système extérieur d'interroger le *noyau* à partir de clauses Prolog[18] : cette vocation est représentée par l'interface LOAD dans le diagramme.

[16]Un diagramme de composants est une structure de l'UML classée parmi les diagrammes d'implémentation. Dans le diagramme en question, le stéréotype "prolog" souligne que ces composants sont implémentés en Prolog.

[17]Un fichier ISL est un fichier représentant des schémas (de données) à l'intention de DB-Main. ISL signifie Information System Specification Language.

[18]La procédure est très simple : le système externe écrit les clauses dans des fichiers et appelle DPT_Load qui charge les clauses nécessaires – y compris celles formant le noyau – dans la base de données de l'interpréteur Prolog.

2. DPT est le coeur du noyau ; il prend en charge les transformations en elle-même, incluant par là les vérifications de validité du diagramme et des préconditions, mais maintenant aussi à jour l'historique des transformations. Il présente, en outre, une série de fonctionnalités orientées vers le confort de programmation telles que de la documentation et des affichages pour correction.
3. TMM est le composant responsable des publications des modèles des transformations. Les interfaces du diagramme représentent les deux fonctionnalités de publication : TransfoList, d'une part, permet à un système externe d'acquérir la liste des transformations disponibles[19] et TransfoModel, d'autre part, peut fournir le modèle d'une transformation[20].
4. CD2ISL est chargé de traduire le diagramme de classes de sa représentation logique vers une représentation externe appelée fichier ISL.

5.4 Intégration dans DB-Main

5.4.1 Voyager 2

Le langage Voyager 2 permet d'écrire un programme qui non seulement interagit avec les données d'un schéma de DB-Main, mais aussi propose une interface homme-machine. Ce programme est appelé un *plug-in* dans DB-Main.

Malheureusement, DB-Main est construit autour du concept entité-relation et non UML. Par conséquent, son méta-modèle de données[21] n'est pas complètement assimilable avec celui que nous manipulons. A cet égard, nous devons donc user de palliatifs : tentant d'utiliser les stéréotypes en dernier ressort, nous nous servirons autant que possible des méta-propriétés (*meta-properties*) de DB-main.

Cette intéressante fonctionnalité de DB-Main matérialisée par les méta-propriétés consiste à définir des propriétés *ad hoc* aux besoins spécifiques de l'utilisateur ; celles-ci concernent un type d'objet précis, tel qu'entité ou attribut[22], et accèptent un type de données paramètrable. Ce type de données peut être choisi parmi une série de types de base (entier, booléen, texte...) ou une liste de ces types de base ; il peut aussi inclure des valeurs prédéfinies ou permettre des valeurs libres.

Nous pourrons donc, par exemple, définir des propriétés "stéréotype", "visibilité" ou encore "paramètres" pour les opérations – plus précisément, leur équivalent dans DB-Main : les *processing units* – qui n'en possédaient pas d'équivalentes dans le schéma standard de DB-Main. La contrepartie de cette adaptation est que notre système ne sera complètement fonctionnel qu'en utilisant un schéma

[19] C'est-à-dire la liste des DP que le système est capable d'utiliser pour transformer le diagramme. Cette information est transmise via un fichier.

[20] Le modèle est transmis via un fichier structuré que le demandeur peut analyser. Ce fichier est organisé en emboitant les informations entre des balises (de manière similaire à un fichier XML).

[21] Détaillé dans Voyager 2 Reference Manual, "Chapter 11 Repository Definition"[V2, p. 63].

[22] Qui représentent respectivement classe et attribut dans notre méta-modèle UML.

spécialisé de DB-Main : celui où nos méta-propriétés particulières ont été introduites.

Nous avons choisi d'ajouter l'ensemble de méta-propriétés suivant pour caractériser les schémas DB-Main que nous utilisons :

1. méta-propriétés des *processing units* (équivalent des opérations dans notre modèle) :
   - VISIBILITY de type liste de texte parmi PRIVATE, PROTECTED et PUBLIC
   - STEREOTYPE de type texte
   - PARAMETERS de type texte structuré comme suit : un paramètre par ligne ; le paramètre est le nom du paramètre ou le nom suivi du type[23] séparé d'un " : ".
2. méta-propriétés des *atomic attributes* (équivalent des attributs dans notre modèle) :
   - VISIBILITY de type liste de texte parmi PRIVATE, PROTECTED et PUBLIC
   - STEREOTYPE de type texte
3. méta-propriété des *entity types* (équivalent des classes dans notre modèle) :
   - STEREOTYPE de type texte

Finalement, le programme écrit en Voyager 2, ou *plug-in*, implémente toute l'interface homme-machine du système si ce n'est le processus de création et de visualisation du diagramme dont se charge déjà DB-Main. Ce *plug-in* a été nommé DPT_Diagram_Export ; il se charge des fonctionnalités suivantes :

1. prendre connaissance du schéma connu de DB-Main et en générer un diagramme de classes sous forme de clauses acceptables par le *noyau ;*
2. questionner le *noyau* pour connaître les transformations possibles et proposer un choix à l'utilisateur ;
3. questionner le *noyau* pour connaître la signature de la transformation choisie et proposer un dialogue[24] avec l'utilisateur pour lui permettre de choisir les paramètres ;
4. transmettre la demande de transformation paramétrée au noyau ;
5. afficher le diagramme transformé[25] à partir du fichier ISL généré par le *noyau.*

5.4.2 Etats du diagramme de classes

Un diagramme de classes prend donc plusieurs formes au fil de sa manipulation par notre système de transformation. Illustrons ce polymorphisme [26] par le diagramme d'état des diagrammes de classes de la figure 5.6.

[23] Parmi les types définis précédemment pour pour les membres en 5.1.

[24] Notons que pour être complètement indépendante de la connaissance des transformations, cette partie nécessite le développement d'un analyseur que nous n'avons pas réellement implémenté.

[25] Un problème dans DB-Main empêche cette fonctionnalité de fonctionner correctement.

[26] Au sens étymologique du terme.

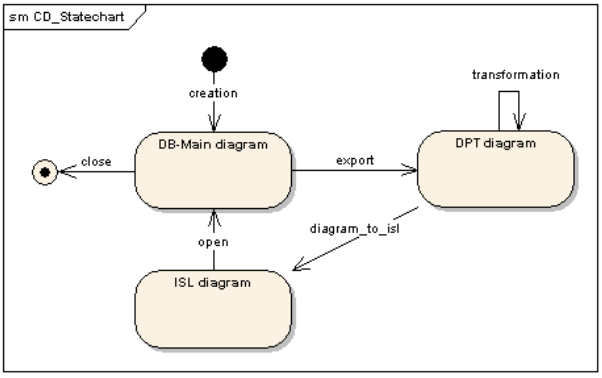

FIG. 5.6: Diagramme d'états des diagrammes de classes

Les états de ce diagramme représentent les différentes formes que prend le diagramme de classes ; les transitions représentent les principales[27] actions du système qui agissent sur le diagramme.

Voici la description des états et des transitions ainsi que des concepts qu'ils recouvrent :

DB-Main diagram : cet état représente la forme objet du diagramme dans la mémoire de travail de DB-Main. Dans cet état, le diagramme est manipulable – et visible – par l'interface utilisateur de DB-Main ainsi que par le biais du *plug-in* DPT_Diagram_Export.

DPT diagram : dans l'état DPT DIAGRAM, le diagramme est sous forme de clauses Prolog conformes au méta-modèle des diagrammes de classes présenté précédemment. Sous cette forme, le diagramme est altérable par le *noyau* et plus précisément par DPT lorsque sont appelées des transformations.

ISL diagram : cet état représente la forme de fichier ISL du diagramme. Sous cette forme, le diagramme n'est pas directement manipulable, c'est une forme de transport vers DB-Main.

creation : la transition CREATION représente la création d'un diagramme par un utilisateur de l'interface de DB-Main.

export : la transition EXPORT représente l'export du diagramme en clauses par le *plug-in* DPT_Diagram_Export.

transformation : la transition TRANSFORMATION représente la transformation du diagramme par une transformation du *noyau*.

diagram_to_isl : la transition DIAGRAM_TO_ISL représente l'export du diagramme sous forme de fichier ISL effectué par le NOYAU (plus précisément par CD2ISL).

[27] Nous aurions pu compléter ce diagramme d'autres transitions et états, mais nous le pensons, au risque de l'alourdir de concepts hors de notre propos. Par exemple, DB-Main permet de sauvegarder un diagramme sous forme de fichier ISL et d'autres formats encore qui auraient pu représenter autant d'états.

open : la transition OPEN représente l'édition du fichier ISL par DB-Main.

close : la transition CLOSE représente la fin de l'utilisation du diagramme.

5.4.3 Composants

Complétons notre description du système de transformation par une vue d'ensemble représentée à l'aide du diagramme de composants de la figure 5.7. Nombre des composants s'y retrouvant ont déjà été décrits : tous ceux du *noyau* (*kernel* dans le diagramme de composants) ainsi que le *plug-in* DPT_Diagram_Export.

Le composant DB-Main représente le logiciel homonyme tandis que le composant stéréotypé "file" et nommé INPUT.LUN représente un éventuel schéma de départ de DB-Main[28].

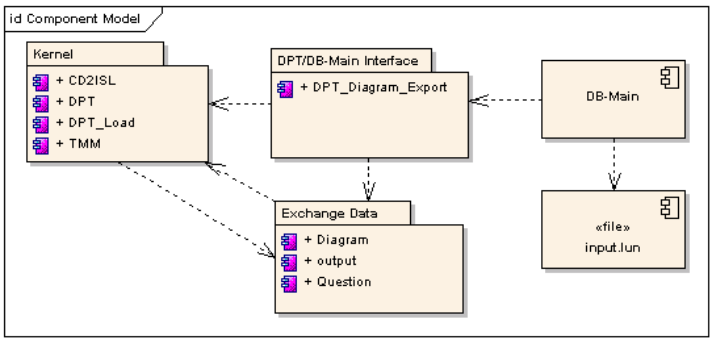

FIG. 5.7: Diagramme de composants – vue d'ensemble

Donc, quelques composant seulement doivent encore être décrits :

Diagram : le composant DIAGRAM représente les clauses Prolog générées par DPT_Diagram_Export dans le but de transmettre la connaissance du diagramme au *noyau*.

Question : le composant QUESTION représente un fichier d'échange Prolog utilisé comme vecteur d'échange d'information entre DPT_Diagram_Export et le *noyau*. Les clauses de ce fichier sont chargées dans la base de données de l'interpréteur Prolog par DPT_Load sous l'impulsion de DPT_Diagram_Export.

output : le composant OUTPUT représente le fichier ISL généré par le noyau après transformation (et selon les indications de QUESTION).

Ayant ainsi modélisé notre système de transformation, nous pouvons nous baser sur les concepts dégagés pour aborder un exemple détaillé d'implémentation d'une transformation.

[28] Au même titre que les fichiers ISL, les fichiers LUN constituent une méthode de sauvegarde des schémas pour DB-Main, mais ils sont plus hardus à créer pour un système externe. LUN signifie Load UNload formated text.

Chapitre 6

Présentation détaillée d'une transformation

Afin d'illustrer un propos encore quelque peu abstrait, nous présentons ci-après une transformation de manière détaillée. La transformation choisie n'est ni la plus simple, ni la plus complexe[1] car nous la souhaitons représentative du concept ; c'est le cas de DÉCORATEUR dont le DP est présenté par ailleurs en 2.2.3.

6.1 Diagramme de classes

Le diagramme de classes à transformer est traduit sous forme de clauses. Ces clauses sont organisées conformément au méta-modèle des classes établi en 5.1.

D'une manière générale, nous ne pouvions choisir de manipuler les éléments du diagramme par leur nom puisqu'un nom peut exister pour plusieurs différents éléments (comme une opération dans deux classes différentes) et que tous les éléments ne sont pas obligatoirement nommés (comme les associations). Donc, nous avons choisi d'utiliser un système numérique en accordant un numéro unique à chaque transformable du diagramme.

En l'occurence, ces numéros suivent la structure suivante :

- les classes sont des multiples de 1000 ;
- les attributs et opérations d'une classe ont pour chiffre des milliers celui de la classe à laquelle ils appartiennent et sont numérotés de 0 à 99 avec le chiffre des centaines à 1 pour les attributs et 2 pour les opérations ;
- les associations (y compris les généralisation selon notre modèle dégradé) sont des multiples de 100000.

Cette méthode limite nos membres à 99 de chaque type par classe et le nombre total de classes d'un diagramme à 99 également. Nous considérons cette limitation comme bénigne au regard des avantages de cette règle : identification

[1] La plus complexe des transformations effectivement implémentée est celle en FABRIQUE ABSTRAITE dont le code peut se retrouver en annexe.

facile de chaque transformable et création aisée d'un nouveau transformable. En effet, créer une classe ne requiert que de trouver le numéro de classe le plus élevé et d'y ajouter 1000 ; créer un membre requiert de trouver le numéro de membre le plus élevé dans la classe et d'y ajouter 1 et créer une association recquiert de trouver le numéro d'association le plus élevé et d'y ajouter 100000.

Un diagramme de classes peut donc être déclaré par l'utilisation de prédicats simples tels que ceux-ci :

class(?Number) : ce prédicat assure ou définit l'existence d'une classe de numéro NUMBER.

class_name(?Number, ?Name) : ce prédicat établit le lien entre un numéro de classe et le nom de cette classe.

attribute(?Number) : ce prédicat assure ou définit l'existence d'un attribut de numéro NUMBER.

member_name(?Number, ?Name) : ce prédicat établit le lien entre un numéro de membre et le nom de ce membre.

has(?ClassNumber, ?MemberNumber) : ce prédicat établit le lien entre une classe et un membre via leurs numéros respectifs.

A ces prédicats doivent s'ajouter ceux nécessaires pour définir des stéréotypes, des visibilités, des opérations, etc. En définitive, tous les concepts du méta-modèle (final) des diagrammes de classes peuvent être représentés de manière similaire.

La première étape d'une procédure d'enrichissement est de porter la connaissance du diagramme de classes parmi les clauses du *noyau* ; c'est la signification de la transition EXPORT du diagramme d'état de la figure 5.6. Cette étape implique de déclarer les connaissances de chaque élément du système en attribuant de nouveaux numéros aux transformables. A cet effet, nous utilisons donc des procédures de création des transformables et de simples clauses déclaratives. Ces procédures de création sont toutes semblables à celles de création d'une classe dont nous donnons l'exemple ici :

assert_class(+ClassName,-ClassNumber) : ce prédicat permet de créer une nouvelle classe dans le diagramme ; il retourne le numéro de la classe créée.

Voici le code Prolog de ce prédicat :

```
assert_class(ClassName,ClassNumber) :-
          new_class(ClassNumber),
          assert(class(ClassNumber)),
          assert(class_name(ClassNumber,ClassName)).
```

Ce code utilise la procédure *new_class/1* définie ci-dessous :

new_class(-NewClass) : ce prédicat permet d'obtenir le prochain numéro de classe non utilisé.

Le code de cette procédure se trouve ci-dessous :

```
new_class(NewC) :-
          class(M),
          max_class(M),
          plus(M,1000,NewC).
%si aucune classe n'existe.
new_class(NewC) :-
          NewC=1000.
max_class(M) :-
          class(X),
          X>M,
          !,
          fail.
max_class(_).
```

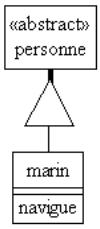

FIG. 6.1: Un diagramme de classes simple

C'est en définissant et en utilisant des procédures similaires à *new_class/1* – quoique souvent plus complexes – que nous pouvons déclarer le diagramme de classes à transformer.

A titre d'exemple, considérons le code de création d'un diagramme de classes très simple correspondant à la figure 6.1 :

```
:-
          assert_class(personne,NewClass),
          assert_class_stereotype(NewClass,abstract).
:-
          assert_class(marin,NewClass),
          assert_operation(NewClass,default_name(navigue),
                    NewOperation1).
:-
          class_name(Parent,personne),
          class_name(Enfant,marin),
          create_generalization(Parent,Enfant,_).
```

Dans ce code, se trouvent trois prédicats encore inconnus : *assert_operation/3* qui permet de créer une nouvelle opération d'une classe, *create_generalization/3* qui permet de créer une nouvelle généralisation entre deux classes et enfin *assert_class_stereotype/2* qui permet d'ajouter un stéréotype à une classe. Pour alléger notre propos, nous avons choisi de ne pas les définir en détail.

6.2 Structure d'une transformation

Une fois le diagramme connu, c'est-à-dire inclus dans la base de données des clauses de l'interpréteur Prolog, la transformation est susceptible d'être appelée.

Définir une transformation passe par cinq étapes de définition : la signature, la précondition, les tâches, les rôles et la documentation.

1. La signature est la définition de l'appel de la transformation ; c'est elle qui détermine quels arguments sont passés. Certains paramètres sont facultatifs et d'autres obligatoires, ce qui en définitive définit plusieurs signatures pour une même transformation à la manière d'une surcharge en programmation orientée objet. Les prédicats qui implémentent la signature se chargent d'abord de contrôler la validité du diagramme, puis de préparer la transformation en découvrant les numéros identifiant les transformables en jeux ; ensuite, ils appellent la transformation en elle-même et finalement gèrent le résultat de celle-ci en informant l'utilisateur de l'échec ou de la réussite. Lors de l'appel de la transformation, la signature ne manque pas de préciser l'absence éventuelle des paramètres facultatifs par l'utilisation du mot-clef "ignored" ou le passage d'une liste vide en cas de paramètre de type liste.
2. La précondition est définie au cas par cas en fonction des besoins spécifiques de la transformation. *A priori*, elle peut concerner tous les faits connus du programme ; en pratique, elle vérifie le plus souvent les qualités des paramètres de la signature (comme le nombre d'éléments d'une liste ou l'abstraction requise d'une classe) avec éventuellement leur rapport à l'historique (comme les rôles qu'ils ont déjà joués dans de précédentes transformations).
3. Les tâches forment le caractère principal de la transformation ; elles altèrent successivement le diagramme afin d'y intégrer le DP en question. Les tâches sont susceptibles de modifier tous les aspects du diagramme ; en pratique, elles ajoutent des éléments ou modifient les qualités des éléments impliqués dans le DP.
4. Avant toute tâche, en début de transformation, mais après vérification des préconditions, est créée la *played_ transformation*. De cette façon, les rôles peuvent être attribués aux éléments impliqués dans la transformation au fur et à mesure de l'application des tâches via l'assertion d'un atome de foncteur *transformed* tel que défini par la relation ternaire de même nom du modèle illustré par la figure 5.2.
5. La documentation décrit la transformation et ses effets, sa précondition et les arguments demandés. De plus, elle définit les rôles qui pourront être attribués.

6.3 Transformation en Décorateur

Cette section propose le code Prolog qui implémente la transformation en DÉCORATEUR. Nous nous sommes volontairement focalisé sur les lignes essentielles, laissant de côté commentaires, procédures de visualisation et de contrôle

visant le confort de programmation, etc. Cette section ne reprend donc pas exactement le code réel.

6.3.1 Documentation

La documentation du DÉCORATEUR ne précise pas le rôle de décorateur concret. Dès lors, ce rôle n'existe pas du point de vue du programme Prolog et les éléments de ce type dans le DP n'en seront pas marqués dans l'historique.

```
doc_transfo(decorator,"Cette transformation crée une
interface décorateur superclasse des classes décorateurs
concrets.",Details) :-
          Details = "1. création d'une classe abstraite
          DécorateurComposant avec les mêmes opérations
          abstraites que son parent.\n 2. ajout d'une
          composition de DécorateurComposant vers Composant.
          \n 3. Création des généralisations des décorateurs
          concrets vers décorateur composant et de
          DécorateurComposant vers Composant.\n 4. Si elles
          n'existent pas, créations des implémentations des
          opérations abstraites de DécorateurComposant dans
          les décorateurs concrets.".
doc_precondition(decorator,"Composant doit être abstrait et
doit avoir au moins une classe spécialisée. Cela signifie qu'
un composant concret au moins doit exister.").
doc_argument(decorator,composant,"nom de la classe
composant").
doc_argument(decorator,decorateurs,"liste des noms des
classes décorateurs concrets").
doc_roles(decorator,[composant,decorateur,decorateurConcret]).
```

6.3.2 Signature

Dans ce cas particulier du DÉCORATEUR, une seule signature est possible.

```
transfo(decorator,[composant(Composant),
decorateurConcrets(L)]) :-
          check_constraints(ConsRes),
          ConsRes = ok,
          prepare_decoratorize(Composant,L,Co,DC),
          !,
          decoratorize(Co,DC),
          check_constraints(_).
transfo(decorator,_) :-
          writef("\ntransformation 'decorator' has been
          called with BAD ARGUMENTS\n",[]),
          fail.
```

La procédure *check_ constraint/1* contrôle la validité du diagramme de classes ; elle est décrite en 6.3.5. Les procédures *prepare_ decoratorize/4* et *decoratorize/2* sont définies ci-dessous.

6.3.3 Tâches et attribution des rôles

Préparation

La préparation telle que décrite en 6.2 est assumée par *prepare_ decoratorize/4*.

```
prepare_decoratorize(Composant,DecoConcret,Co,DC) :-
          class(Co),
          class_name(Co,Composant),
          term_in_names(DecoConcret,TempL,decorateurConcret),
          class_list(TempL,DC).
```

La procédure *term_ in_ names(+ListIn,-ListOut,+TermName)* transforme une liste de termes identiques d'arité 1 en la liste des arguments des termes tandis que *class_ list(+NameList,-ClassList)* fournit la liste des numéros des classes dont la liste des noms a été passée.

Transformation

La procédure *decoratorize/2* effectue les tâches et attribue les rôles de la transformation en DÉCORATEUR après avoir vérifié les préconditions. Les tâches effectuées sont :

1. créer la classe abstraite décorateur en y ajoutant des opérations clonant les opérations abstraites de composant ;
2. créer une composition nommée *composant* de la classe décorateur vers la classe composant ;
3. créer les généralisations des décorateurs concrets vers le décorateur et de celui-ci vers la classe composant ;
4. si elles n'existent pas déjà, créer des implémentations[2] des opérations abstraites de la classe décorateur dans les décorateurs concrets.

```
decoratorize(Composant,DecorateursConcrets) :-
          precondition(decorator,[Composant,
          DecorateursConcrets]),
          !,
          add_played_transfo(decorator),
          %1. création d'une classe abstraite Décorateur
          create_decorateurComposant(Composant,
          DecorateurComposant),
          add_transformed(DecorateurComposant,decorateur),
          %2. ajout de la composition
```

[2]Nous entendons par là des copies des opérations, mais sans le stéréotype "abstract".

```
            create_composition(DecorateurComposant,Composant,
            Composition),
            assert(association_name(Composition,composant)),
            add_transformed(Composant,composant),
            %3. création des généralisations
            create_generalization(Composant,
            DecorateurComposant,_),
            generalize_list(DecorateurComposant,
            DecorateursConcrets,[],_),
            add_transformed_list(DecorateursConcrets,
            decorateurConcret),
            %4. créations des implémentations des opérations
            implement_abstract_ope(DecorateurComposant,
            DecorateursConcrets,[]).
decoratorize(_,_,_) :-
            transfo_fail.
```

Voici une description succinte des nouvelles procédures : *create_ composition/3* permet la création d'une nouvelle relation de composition entre deux classes ; *generalize_ list/4* permet la création d'une relation de spécialisation entre une classe et toutes les classes d'une liste de classes ; *implement_ abstract_ ope/3* permet la création d'opérations de même nom que les opérations abstraites d'une classe vers toutes les classes d'une liste en omettant le stéréotype "abstract" des opérations clonées.

Les procédures *precondition/2* et *create_ decorateurComposant/2* ainsi que les procédures *add_ played_ transfo/1*, *add_ transformed/2* et *add_ transformed_ list/3* sont décrites ci-après tandis que *create_ generalization/3* a déjà été évoquée en 6.1.

Historique

La transformation participe à l'historique lorsqu'elle est concrétisée. Pour ce faire, elle utilise les procédures dont voici la définition :

```
last_played_transfo_num(0).
played_transfo(0,initialisation).
transformed(0,0,initialisation).
add_played_transfo(Transfo) :-
            doc_transfo(Transfo,_,_),
            new_num(Num),
            assert(played_transfo(Num,Transfo)).
new_num(Num) :-
            last_played_transfo_num(I),
            plus(I,1,Num),
            retract(last_played_transfo_num(I)),
            assert(last_played_transfo_num(Num)).
add_transformed(Transformable,Role) :-
            transformable(Transformable),
```

```
            last_played_transfo_num(PTNum),
            role_exist(PTNum,Role),
            assert(transformed(Transformable,PTNum,Role)).
role_exist(PTN,Role) :-
            played_transfo(PTN,Transfo),
            doc_roles(Transfo,L),
            member(Role,L).
has_role(Tr,Role,Transfo) :-
            has_role(Tr,Role,Transfo,_).
has_role(Tr,Role,Transfo,PTNum) :-
            transformed(Tr,PTNum,Role),
played_transfo(PTNum,Transfo).
add_transformed_list([H|T],Role) :-
            add_transformed(H,Role),
            add_transformed_list(T,Role).
            add_transformed_list([],_).
```

Les procédures suivantes jouent un rôle important dans le maintien de l'historique : *add_ played_ transfo/1* définit une *played_ transformation* et lui attribue un numéro unique ; *add_ transformed/2* permet d'associer un transformable et un rôle via un atome de foncteur *transformed* ; *add_ transformed_ list/3* permet d'associer les transformables d'une liste à leur même rôle.

6.3.4 Précondition

Les préconditions d'un décorateur impliquent de vérifier que la classe passée comme composant est abstraite et qu'elle est héritée d'au moins une classe, c'est-à-dire qu'un composant concret existe.

Nous aurions, bien sûr, pu définir d'autres préconditions pour le même DP en fonction de nos recherches et de notre désir d'assouplir ou au contraire de mieux circonscrire les possibilités de transformation, mais nous avons jugé cette précondition suffisante.

```
precondition(decorator,[Composant,DecorateursConcrets]) :-
            class_stereotype(Composant,abstract),
            generalization_exists(Composant,_,_).
precondition(decorator,_) :-
            writef("\nPreconditions not fullfill."),
            !,
            fail.
```

La procédure *generalization_ exist/3* vérifie qu'une classe est spécialisée par un autre.

6.3.5 Procédures utilitaires

Procédure sur mesure

Certaines procédures utilitaires sont construites sur mesure afin de permettre une transformation particulière ; c'est le cas de *create_ decorateurComposant/2* qui permet de créer une classe de rôle décorateur dont le nom est la concaténation de "decorator" et du nom de la classe de rôle composant (en suivant ce nom d'un chiffre si le nom est déjà utilisé).

```
create_decorateurComposant(Composant,DecorateurComposant) :-
          class_name(Composant,ComposantName),
          atom_concat(decorator,ComposantName,DCName),
          copy_abstract_class(Composant,DecorateurComposant,
          default_name(DCName)).
```

La procédure *copy_ abstract_ class/3* permet de cloner une classe en n'en copiant que les opérations stéréotypées "abstract".

Procédure générale

Le programme Prolog formant le *noyau* comprend un grand nombre de procédures utilitaires ; nous ne pouvons en fournir la liste complète, trop longue pour les exigences du genre. A titre d'exemple, nous fournissons la procédure *check_ diagram/1* qui contrôle la validité du diagramme de classes. Comme la procédure entière (y compris ses sous-procédures) compte plus de cent-cinquante lignes, nous n'en donnons que le début et invitons le lecteur curieux à consulter l'annexe.

La procédure vérifie que les classes ont des noms différents, que les attributs d'une même classe ont des noms différents, que les opérations d'une même classe ont des signatures différentes et que les paramètres d'une même opération ont des noms différents et des types acceptés.

```
check_constraints(Res) :-
          check_result_ok,
          all_class_distinct([]),
          check_result(Res,L).
check_constraints(Res) :-
          check_result(Res,L),
          write_res(Res,L).
check_result_ok :-
          remove_check_result,
          assert(check_result(ok,[])).
remove_check_result :-
          check_result(A,B),
          !,
          retract(check_result(A,B)).
remove_check_result.
```

La procédure *all_ class_ distinct* procède aux vérifications attendues (unicité des noms de classes, mais aussi des noms des attributs des classes et des signatures des opérations des classes). En cas de non validité, elle change l'assertion de *check_ result/2* en y passant les valeurs qui permettront de décrire les causes du problème. La procédure *write_ res/2* se charge alors de rendre ce résultat intelligible à l'utilisateur en affichant un texte descriptif.

6.4 Publication et appel

Afin d'être utilisable, une transformation doit encore publier sa signature comme défini en 5.2.2. Cette publication passe par l'impression dans un fichier des caractéristiques de la transformation. Voici les faits qui définissent cette signature pour la transformation en DÉCORATEUR :

```
signature(decorator,parameters([Arg1,Arg2])) :-
          Arg1 = parameter(composant,faux,[class_name]),
          Arg2 = parameter(decorateurConcrets,faux,[list
          (parameter(decorateurConcret,[class_name]),1)]).
```

Et voici le contenu du fichier généré pour spécifier la signature de la transformation en DÉCORATEUR :

```
<signature>
     <transfoName>decorator</transfoName>
     <parameters>
          <parameter>
               <parameterName>composant</parameterName>
               <facultative>faux</facultative>
               <parameterType>class_name</parameterType>
          </parameter>
          <parameter>
               <parameterName>decorateurConcrets
               </parameterName>
               <facultative>faux</facultative>
               <listParameterType>
                    <min>1</min>
                    <parameter>
                         <parameterName>decorateurConcret
                         </parameterName>
                         <parameterType>class_name
                         </parameterType>
                    </parameter>
               </listParameterType>
          </parameter>
     </parameters>
</signature>
```

Ces informations signifient en somme que la transformation en DÉCORATEUR s'appelle *decorator* et demande deux paramètres : le premier nommé *compo-*

sant doit être un nom de classe et est obligatoire ; le second nommé *decorateurConcrets* est lui aussi obligatoire et est constitué d'une liste d'au moins un paramètre nommé *decorateurConcret* qui doit être un nom de classe.

A titre d'exemple, si un utilisateur souhaite appeler la transformation en DÉCORATEUR sur le diagramme de classes défini à la figure 6.1, il utilisera le but Prolog suivant :

```
:-
        transfo(decorator,[composant(personne),
        decorateurConcrets([decorateurConcret(marin)])]),
        diagram_to_isl.
```

La procédure *diagram_to_isl/0* prend en charge la création du fichier ISL.

Chapitre 7

Liste de transformations

A titre de preuve de concept, nous avons développé cinq transformations en autant de DP : Singleton, Composite, Stratégie, Décorateur et Fabrique Abstraite.

7.1 Transformation en Singleton

7.1.1 Rôles

Les rôles choisis pour identifier les éléments importants d'un SINGLETON sont :

- *singleton* qui identifie la classe singleton,
- *uniqueInstance* qui identifie l'attribut unique instance,
- *instance* qui identifie l'opération instance,
- *constructeur* qui identifie le constructeur.

7.1.2 Signature

Appeler la transformation en SINGLETON demande un paramètre obligatoire :

- le *singleton* qui doit être un littéral représentant le nom d'une classe.

7.1.3 Précondition

La précondition de la transformation en SINGLETON consiste en ceci :

- la classe à transformer ne doit pas déjà être un singleton.

7.1.4 Tâches

Les tâches de la transformation en SINGLETON sont :

1. assurer que le constructeur existe et lui donner la visibilité protected ;
2. ajouter l'attribut uniqueInstance ;
3. ajouter l'opération instance.

7.2 Transformation en Composite

7.2.1 Rôles

Les rôles choisis pour identifier les éléments importants d'un COMPOSITE sont :

- *composant* qui identifie la classe composant,
- *composite* qui identifie le composite,
- *feuille* qui identifie les classes feuilles,
- *client* qui identifie l'éventuel client.

7.2.2 Signature

Appeler la transformation en COMPOSITE demande un paramètre obligatoire et d'autres facultatifs :

Le paramètre obligatoire est :

- la *liste des feuilles* qui doit être une liste non vide de termes *feuille/1* dont l'argument est le nom de la classe feuille.

Les paramètres facultatifs sont :

- l*e composite* qui doit être un littéral représentant le nom d'une classe,
- le *composant,* un littéral représentant le nom d'une classe,
- le *client*, un littéral représentant le nom d'une classe.

Notons que soit le paramètre composant, soit le paramètre composite doit être présent, ou les deux.

7.2.3 Précondition

La précondition de la transformation en COMPOSITE consiste en ceci :

- les classes composant et composite ne peuvent déjà avoir les rôles de composant et composite dans une même *played_ transformation* de type composite précédemment appliquée.

7.2.4 Tâches

Les tâches de la transformation en COMPOSITE sont :

1. si le composant n'a pas été passé (paramètre facultatif), le créer à partir d'un clone de composite ; ensuite, rendre abstrait le composant et ses opérations et y ajouter après les avoir créées les opérations add, remove et getChild ;
2. si le composite n'a pas été passé (paramètre facultatif), le créer à partir d'un clone rendu concret de composant ; ensuite, ajouter des opérations add, remove et getChild dans composite si elles n'y sont pas déjà et finalement ajouter à composite une note explicative de l'implémentation des opérations héritées de composant ;

3. si elles n'existent pas encore, créer les généralisations des feuilles et de composite vers composant ;
4. créer la composition nommée *children* de composite vers composant ;
5. si un client a été passé (paramètre facultatif), créer une association link entre client et composant.

7.3 Transformation en Stratégie

7.3.1 Rôles

Les rôles choisis pour identifier les éléments importants d'une Stratégie sont :

– *context* qui identifie la classe contexte,
– *strategy* qui identifie la classe stratégie,
– *concreteStrategy* qui identifie les classes stratégies concrètes.

7.3.2 Signature

Appeler la transformation en Stratégie demande trois paramètres obligatoires :

– le *contexte* qui doit être le nom d'une classe ;
– la *strategie abstraite*, le nom d'une classe ;
– la *liste des stratégies concrètes* qui doit être une liste non vide de termes *concreteStrategy/1* dont l'argument est le nom de la classe stratégie concrète.

7.3.3 Précondition

Nous n'avons pas déterminé de précondition spécifique à la transformation en Stratégie, jugeant que le cadre imposé par la signature était suffisant. Dès lors, la vérification de la précondition de cette transformation est toujours vraie.

7.3.4 Tâches

Les tâches de la transformation en Stratégie sont :

1. si elle ne l'est pas déjà, rendre la classe stratégie abstraite ;
2. si elles n'existent pas encore, créer les généralisations de chaque stratégie concrète vers la super-classe stratégie ;
3. créer la composition nommée *strategy* de contexte vers stratégie.

7.4 Transformation en Décorateur

La transformation en Décorateur a déjà été décrite en détail au point 6.3.

7.5 Transformation en Fabrique Abstraite

7.5.1 Rôles

Les rôles choisis pour identifier les éléments importants d'une FABRIQUE ABSTRAITE sont :

- *abstractFactory* qui identifie la classe fabrique abstraite,
- *concreteFactory* qui identifie les classes fabriques concrètes,
- *client* qui identifie la classe client,
- *abstractProduct* qui identifie les classes produits abstraits,
- *concreteProduct* qui identifie les classes produits concrets.

7.5.2 Signature

Appeler la transformation en Fabrique Abstraite demande des paramètres obligatoires et d'autres, facultatifs.

Les paramètres obligatoires sont :

- la *fabrique abstraite* qui doit être un littéral représentant le nom d'une classe ;
- la *liste des familles de produits* qui doit être une liste de littéraux représentant des familles de produits ;
- le *client* qui doit être un nom de classe ;
- la *liste des produits* qui doit être une liste de termes *produit*/2 dont le premier argument est un nom de classe représentant une classe produit abstrait et le second argument est la liste, éventuellement vide[1], des noms des classes produits concrets (fournis dans le même ordre que celui des familles auxquelles ces produits concrets correspondent).

Le paramètre facultatif est :

- la liste des fabriques concrètes qui doit être une liste de littéraux représentant les noms des classes qui joueront le rôle de famille concrète[2].

7.5.3 Précondition

La précondition de la transformation en Fabrique Abstraite consiste en ceci :

- la fabrique abstraite doit être abstraite[3],
- le nombre de fabriques concrètes passées doit être inférieur ou égal au nombre de familles,
- pour chaque produit, le nombre de produits concrets doit être inférieur ou égal au nombre de familles.

[1] Donc, le sous-paramètre *liste des produits concrets* pourrait être considéré comme un paramètre facultatif.

[2] Ces familles concrètes doivent encore être passées dans l'ordre les familles de produits.

[3] Le livre Design Pattern[GoFDP] décrit un exemple où la fabrique abstraite est une des implémentations concrètes. Nous ne considérons pas ce cas.

7.5.4 Tâches

Les tâches de la transformation en Fabrique Abstraite sont :

1. créer une association link de client vers la classe fabrique abstraite et vers chaque produit abstrait ;
2. assurer que chaque famille ait sa fabrique concrète, donc,
   a) pour chaque famille concrète passée, vérifier qu'elle possède l'implémentation des méthodes abstraites de la classe fabrique abstraite, sinon les créer et,
   b) pour chaque famille au-delà des fabriques concrètes passées (toutes si aucune fabrique concrète n'a été passée), créer une fabrique concrète implémentant toutes les méthodes abstraites de la classe fabrique abstraite ;
3. si elles n'existent déjà, créer les généralisations de chaque fabrique concrète vers la classe fabrique abstraite ;
4. pour chaque produit, vérifier que tous les produits concrets existent ; donc, vérifier que chacun de ces produits concrets passés implémente les méthodes abstraites du produit abstrait, sinon, créer les méthodes.

Conclusion

Nous avons donc décrit et implémenté cinq transformations de diagrammes de classes à l'aide de DP. Celles-ci sont le résultat de bien des choix de conception dont nous discutons dans le chapitre 9.

Considérant que chacune de ces transformations puisse favorablement altérer un diagramme de classe, leur intérêt resterait pourtant limité si elles ne pouvaient également être combinées pour en enrichir un.

Dès lors, nous complètons la description de ce système de transformations par une étude de cas qui utilise chacune des cinq transformations mises au point.

Quatrième partie

Etude de cas

Chapitre 8

Enrichissement d'un diagramme de compilateur

La présentation de notre système de transformation ne suffit probablement pas à suggérer une idée précise de ses capacités, de ses avantages comme de ses limitations. Nous proposons donc un exemple qui combine les cinq transformations dont est capable notre solution.

A cette fin, nous avons imaginé un problème dont le but était, *in fine*, de pouvoir être solutionné par l'utilisation des cinq DP concernés.

8.1 Problème

Le problème est celui-ci : nous souhaitons concevoir un compilateur de programmes pour une machine-outil difficilement accessible. Ce compilateur sera intégré dans une application existante qui utilisera ses capacités par l'intermédiaire d'une classe à définir.

Supposons donc qu'une analyse préliminaire ait mis en avant quelques instructions de base des programmes ; soient "chargement", "déchargement" et "destruction" – de nouvelles instructions sont susceptibles d'être ajoutées si le projet est un succès. Ces instructions de base seront combinées par les programmeurs en utilisant des structures conditionnelles et des sauts entre des groupes d'instructions. L'analyse a permis de mettre au point une génération de code pour chacune de ces instructions. Notre compilateur devra donc être capable de lire le programme qui les utilise en les groupant en bloc d'instructions et d'en générer le code destination dans l'ordre adéquat selon une série de règles de préséance connues.

En outre, ces instructions de bases pourront être aggrémentées d'ordres de travaux annexes que la machine-outil – par exemple, une grue automatisée – peut réaliser par ailleurs : "pesée", "comptage" et "tamisage" ont d'ores et déjà été identifiés comme tels. Par exemple, une instruction "chargement" pourrait être agrémentée d'une "pesée" et d'un "tamisage" ce qui signifierait que lors du chargement, la machine devrait peser et tamiser sa charge.

De plus, la machine-outil est imposante, lointaine et chère. Des programmeurs devront utiliser notre compilateur pour créer des applications qui dirigeront la machine selon certains scénarios préétablis. Ces informaticiens ne disposeront pas de copie de son système informatique ; donc, ils devront être capables de tester leurs programmes en simulation avant une mise en production coûteuse et difficile. Nous supposons le simulateur fonctionnel, mais utilisant un jeu d'instructions différent de celui de la machine-outil.

Enfin, les applications seront programmées conformément à des scénarios mis au point par des spécialistes du domaine, néanmoins faillibles face à certaines optimisations possibles des procédures. Un système expert existant pourra donc analyser leur code en profondeur afin d'optimiser le travail de la machine selon une série de critères variés – par exemple, coût en énergie, sécurité du personnel ou règlementation en vigueur. Néanmoins, cette optimisation très coûteuse en temps ne sera utilisée qu'une fois le programme fonctionnellement mis au point et testé en simulation.

De cette définition du problème, nous pouvons déduire les besoins suivants :

1. une classe facilement utilisable par un programme engloblant ;
2. une structure d'instructions de base et de blocs d'instructions qui génèrent leur propre code ;
3. pouvoir ajouter des responsabilités aux instructions de base :
4. pouvoir générer deux types de code compilé ;
5. pouvoir définir deux façons de générer le code à partir de la structure.

8.2 Solution

Bien sûr, la solution à tous nos besoins est fournie par les cinq DP prévus.

D'abord, une classe compilateur est définie pour gérer l'ensemble du processus ; elle doit être unique et accessible par le reste du système. C'est le Singleton qui est utilisé à cette fin[1].

Ensuite, une structure arborescente est idéale pour décrire nos instructions[2] : c'est Composite qui la définit.

L'ajout des responsabilités aux instructions de base est, lui, assuré par l'utilisation de Décorateur.

De plus, la Fabrique Abstraite permet de créer une structure capable de générer un code destination différent pour la machine réelle et pour le simulateur.

Enfin, les deux façons de générer le code sont clairement différenciées grâce au DP Stratégie.

[1] Bien que dépassant notre propos, précisons que la classe compilateur constitue une Façade pour le reste du système selon le DP ainsi nommé défini *in Design Patterns* [GoFDP, p. 215].

[2] Représentant, *mutatis mutandis*, un arbre syntaxique.

8.2.1 Diagramme de départ

La description du problème nous permet d'identifier les classes du diagramme de départ que nous allons enrichir par la suite. Ces classes et leurs membres, paramètres de base des transformations, sont représentés dans le diagramme de classes à la figure 8.1.

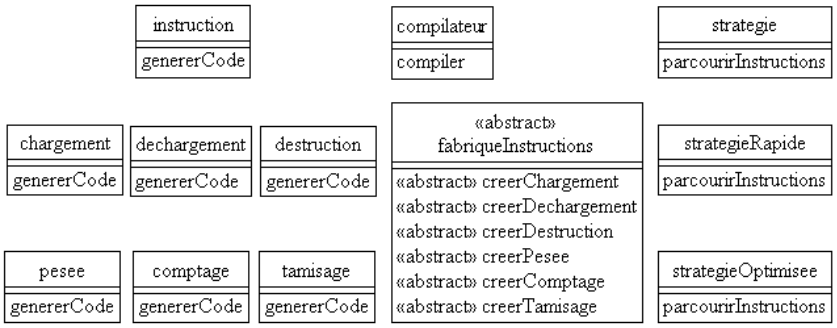

FIG. 8.1: Diagramme de classes du compilateur – départ

Dans ce diagramme, apparaissent les classes qui préfigurent notre solution. Ainsi, COMPILATEUR est la classe centrale destinée à utiliser toutes les autres ; INSTRUCTION sera une super-classe pour les instructions de base qui pourront générer leur propre code destination par une opération GENERERCODE. Ces instructions de bases sont représentées par les classes CHARGEMENT, DECHARGEMENT et DESTRUCTION. Nous devons aussi ajouter les instructions additionnelles sous forme des classes PESEE, COMPTAGE et TAMISAGE qui, elles-aussi, ont une opération GENERERCODE.

Ces classes représentant des instructions seront organisées en une structure hiérarchique parcourue selon différentes stratégies : celles-ci sont représentées par une classe générale STRATEGIE et ses sous-classes STRATEGIERAPIDE et STRATEGIEOPTIMISEE, lesquelles trois classes ont une opération PARCOURIRINSTRUCTIONS qui servira à parcourir la structure des instructions en lui demandant de générer son code, le parcours s'effectuant de différentes manières selon qu'on utilise telle ou telle stratégie. On suppose que STRATEGIEOPTIMISEE utilisera le système expert plus lent tandis que STRATEGIERAPIDE utilisera un type de parcours plus simple, défini intrinsèquement.

Enfin, la classe abstraite FABRIQUEINSTRUCTIONS[3] est présentie pour définir des fabriques d'instructions, c'est-à-dire des classes qui permettront de créer des instances d'instructions qui génèrent le code destination tantôt pour le simulateur, tantôt pour la machine, selon le choix de l'utilisateur du compilateur.

[3]Comme DB-Main et notre système ne proposent pas d'élément *interface*, nous stéréotypons une classe de "abstract" pour jouer ce rôle et l'appelons classe abstraite ; chacune de ses opérations reçoit également le stéréotype "abstract".

8.3 Transformations

Les transformations sont contrôlées par une interface graphique dont nous montrons certains écrans. Notre illustration est surtout supportée par les diagrammes résultant des transformations successives. En effet, les contraintes du genre ne nous permettent pas une illustration exhaustive de l'étude de cas et nous avons dû nous limiter aux images les plus exemplatives[4].

Afin de proposer une description claire, nous avons retouché les diagrammes résultant des transformations, mais en tentant de ne pas altérer la démonstration ou cacher des imperfections.

En effet, notre système ne gère pas les positionnements des éléments des diagrammes et la fonctionnalité de placement automatique de DB-Main ne place pas les éléments par groupe logique. Nous avons donc manuellement placé les éléments du diagramme.

De plus, notre système ne peut générer des compositions dans DB-Main lors de la phase d'export du diagramme vers le fichier ISL[5] bien qu'il les traite au niveau du *noyau*. Cet export se contente alternativement de stéréotyper la relation de "cmp" pour signifier la composition. Nous avons manuellement créé les compositions ainsi identifiées dans les diagrammes.

Ensuite, le diagramme produit est par défaut sous sa forme entité-association plus naturelle à DB-Main ; nous utilisons donc la fonctionnalité de visualisation en diagramme UML de DB-Main.

Finalement, d'autres détails ne sont pas idéalement gérés et impliquent quelques manipulations que nous indiquons dans la description des transformations qui suit.

8.3.1 Transformation par Singleton

La première transformation choisie est celle de la classe COMPILATEUR en SINGLETON.

Après le lancement du plug-in DPT_Diagram_Export[6], l'interface utilisateur propose de choisir une transformation comme illustré en la figure 8.2. A la suite du choix de la transformation en SINGLETON, le système propose de choisir la classe qui jouera le rôle de singleton dans le DP par le biais de l'écran représenté par la figure 8.3.

Ce processus de choix de la transformation suivie du choix des paramètres se répètera pour chaque transformation, souvent avec plus de paramètres que pour le cas du singleton, mais toujours de manière similaire par un écran interactif. C'est pourquoi, nous n'illustrons pas ce processus pour chaque transformation par les figures.

[4] Toutefois, le présent texte est complété d'un vidéogramme démonstratif de notre étude de cas.

[5] Cette phase correspond à la transition diagram_to_isl décrite au point 5.4.2.

[6] DPT_Diagram_Export est décrit en 5.4.1.

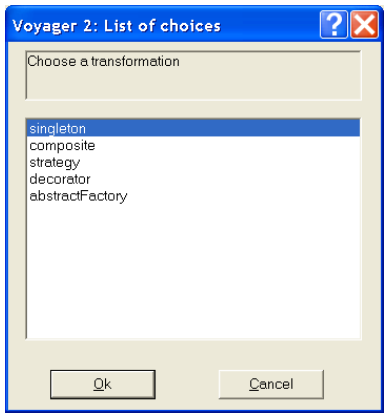

FIG. 8.2: Choix de la transformation en Singleton

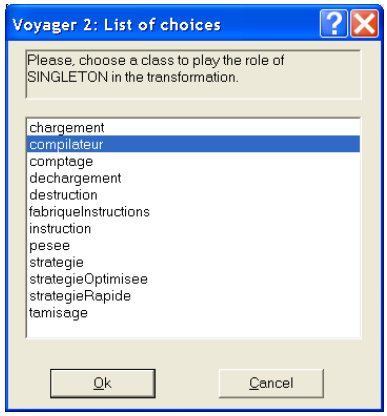

FIG. 8.3: Choix de la classe Singleton

Finalement, la transformation produit un diagramme qui, après quelques réarrangements, est montré à la figure 8.4. Conformément à notre attente, la classe compilateur y est devenue un SINGLETON, susceptible d'être accédée facilement par le reste du système via son opération INSTANCE[7]. Ce diagramme sert de base à la prochaine transformation par le DP COMPOSITE.

8.3.2 Transformation par Composite

La seconde transformation est la transformation en COMPOSITE.

Nous choisissons d'attribuer le rôle de COMPOSANT à la classe INSTRUCTION et d'ignorer le rôle de COMPOSITE comme paramètre facultatif. Les FEUILLES

[7]Bien que ce soit invisible dans l'illustration, INSTANCE est bien une opération publique (*public*) tandis que l'atribut UNIQUEINSTANCE est privé (*private*) et le constructeur COMPILATEUR protégé (*protected*).

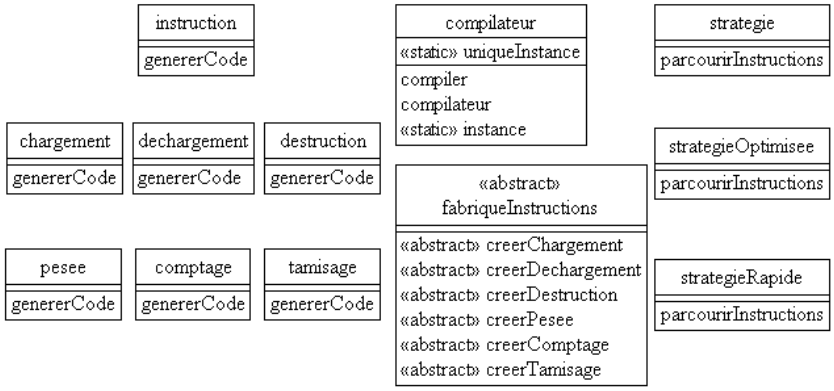

FIG. 8.4: Transformation du compilateur par Singleton

choisies sont CHARGEMENT, DESTRUCTION et DECHARGEMENT et le CLIENT, la classe COMPILATEUR.

Après transformation et réarrangement, le système produit le diagramme illustré par la figure 8.5.

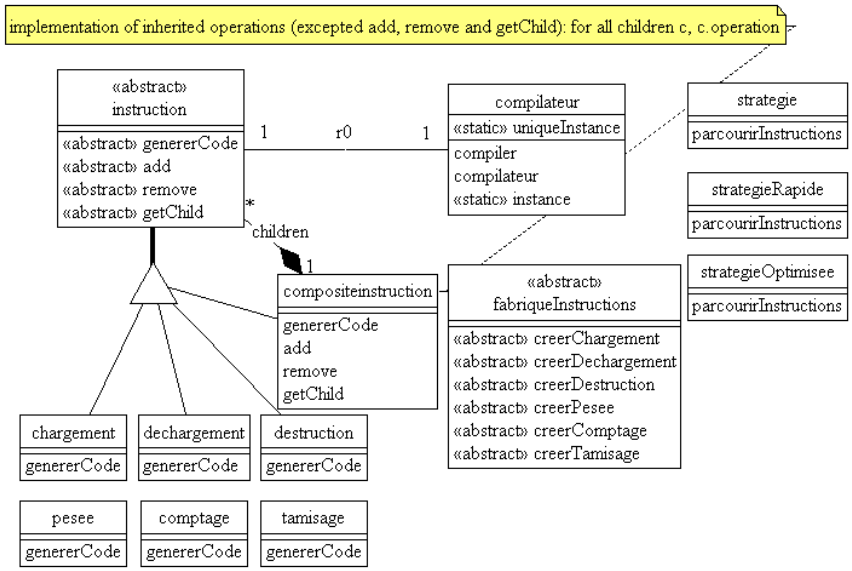

FIG. 8.5: Transformation du compilateur par Composite

Ce diagramme montre une nouvelle classe COMPOSITEINSTRUCTION, c'est la classe de rôle COMPOSITE qui permettra de créer une structure hiérarchique des instructions en définissant des blocs d'instructions par sa composition CHILDREN. Elle est accompagnée d'une note décrivant le conseil d'implémentation de son opération GENERERCODE qui doit appeler les opérations de même nom

de ses composants (CHILDREN). Les classes auxquelles nous avons attribué les rôles et cette nouvelle classe forment un DP COMPOSITE.

8.3.3 Transformation par Décorateur

Le diagramme représenté par la figure 8.5 est le diagramme de base du prochain enrichissement par le DP DÉCORATEUR.

La transformation prend ici pour paramètre composant la classe INSTRUCTION et pour décorateurs concrets PESEE, COMPTAGE et TAMISAGE. Elle produit le diagramme illustré par la figure 8.6.

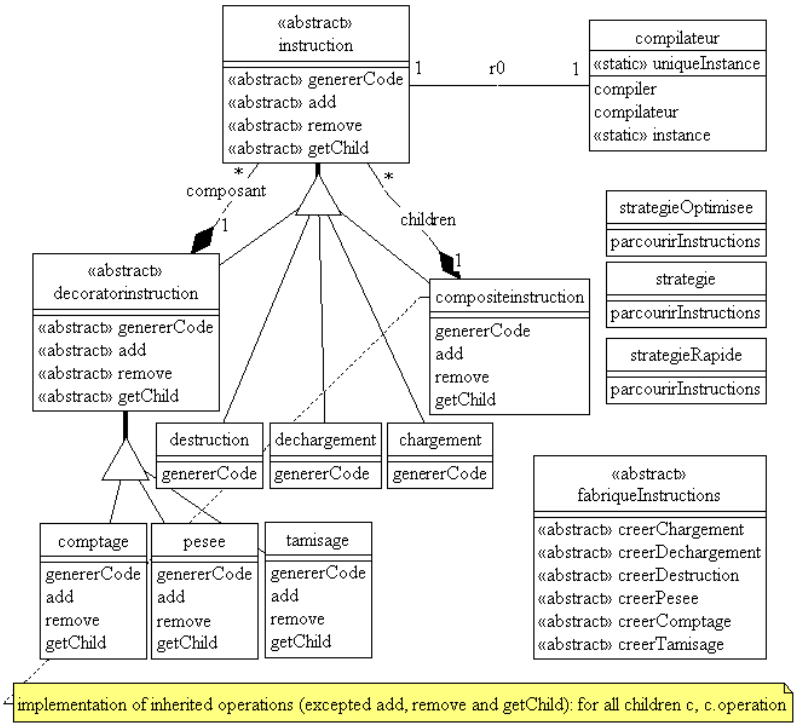

FIG. 8.6: Transformation du compilateur par Décorateur

Notons que notre système perd la note lors de l'export de DB-Main vers le *noyau*[8]. Sa présence est un artifice obtenu manuellement.

Le diagramme obtenu présente à présent une certaine complexité. En effet, la classe instruction y joue à la fois le rôle de COMPOSANT du DP COMPOSITE et COMPOSANT du DP DÉCORATEUR. En outre, la transformation par ce DP a créé les opérations ADD, REMOVE et GETCHILD dans les DÉCORATEURS CONCRETS leur permettant par là de décorer n'importe quelle instruction.

[8]Représenté par la transition EXPORT en 5.4.2.

Pourtant, DÉCORATEUR se distingue facilement dans la structure en considérant DESTRUCTION, DECHARGEMENT et CHARGEMENT (voire COMPOSITEINSTRUCTION) comme des COMPOSANTS CONCRETS.

A l'aide de ce DP, nous sommes à présent en mesure d'attribuer aux instructions de base les responsabilités suplémentaires induites par COMPTAGE, PESEE et TAMISAGE. Il suffit à cet égard d'en décorer les instructions car les implémentations de GENERERCODE de ces DÉCORATEURS CONCRETS doivent appeler l'opération homonyme de leur composant (l'instruction qu'elles décorent) en y ajoutant le code spécifique à leur fonction (respectivement le comptage, la pesée et le tamisage).

8.3.4 Transformation par Fabrique Abstraite

Le quatrième enrichissement est réalisé par la transformation à l'aide du DP FABRIQUE ABSTRAITE.

Les arguments choisis pour cette transformation sont : FABRIQUEINSTRUCTION pour le rôle de FABRIQUE ABSTRAITE, COMPILATEUR pour le rôle de CLIENT et les six classes CHARGEMENT, DESTRUCTION, DECHARGEMENT, PESEE, COMPTAGE et TAMISAGE pour les PRODUITS ABSTRAIT. Aucun PRODUIT CONCRET ni FABRIQUE CONCRÈTE n'a été passé et les familles[9] de produits entrées sont "simulateur" et "machine".

Ces arguments ont permis à la transformation de mener le diagramme à celui présenté à la figure 8.7. Comme CLIENT de la FABRIQUE ABSTRAITE, COMPILATEUR peut maintenant utiliser la FABRIQUE CONCRÈTE adéquate pour créer les instructions[10] de sa structure d'instructions qui soient capables de générer du code destination pour le simulateur ou pour la machine[11], en supposant que les douze PRODUITS CONCRETS créés en soient capables.

Notons par ailleurs que DB-Main prend la liberté de transformer deux relations binaires en une relation ternaire pendant notre processus – les relations entre INSTRUCTION et COMPILATEUR ainsi qu'entre COMPILATEUR et FABRIQUEINSTRUCTION – ce qui n'est pas adéquat puisque nous utilisons les relations pour signifier des associations en UML. Nous devons donc recourir à une manipulation pour obtenir le diagramme souhaité.

8.3.5 Transformation par Stratégie

La dernière transformation est la transformation en STRATÉGIE.

Nous choisissons d'attribuer le rôle de STRATÉGIE à la classe STRATEGIE et celui de STRATÉGIE CONCRÈTE à STRATEGIERAPIDE et STRATEGIEOPTIMISEE tandis que COMPILATEUR tient le rôle de CONTEXTE.

[9] Voir en 7.5.2 pour la définition de cet argument.

[10] Exceptée COMPOSITEINSTRUCTION.

[11] Par exemple, si COMPILATEUR utilise la FABRIQUE CONCRÈTE SIMULATEURFABRIQUEINSTRUCTIONS pour créer un TAMISAGE via l'opération CREERTAMISAGE, il recevra une instance de SIMULATEURTAMISAGE capable de générer du code destination pour le simulateur.

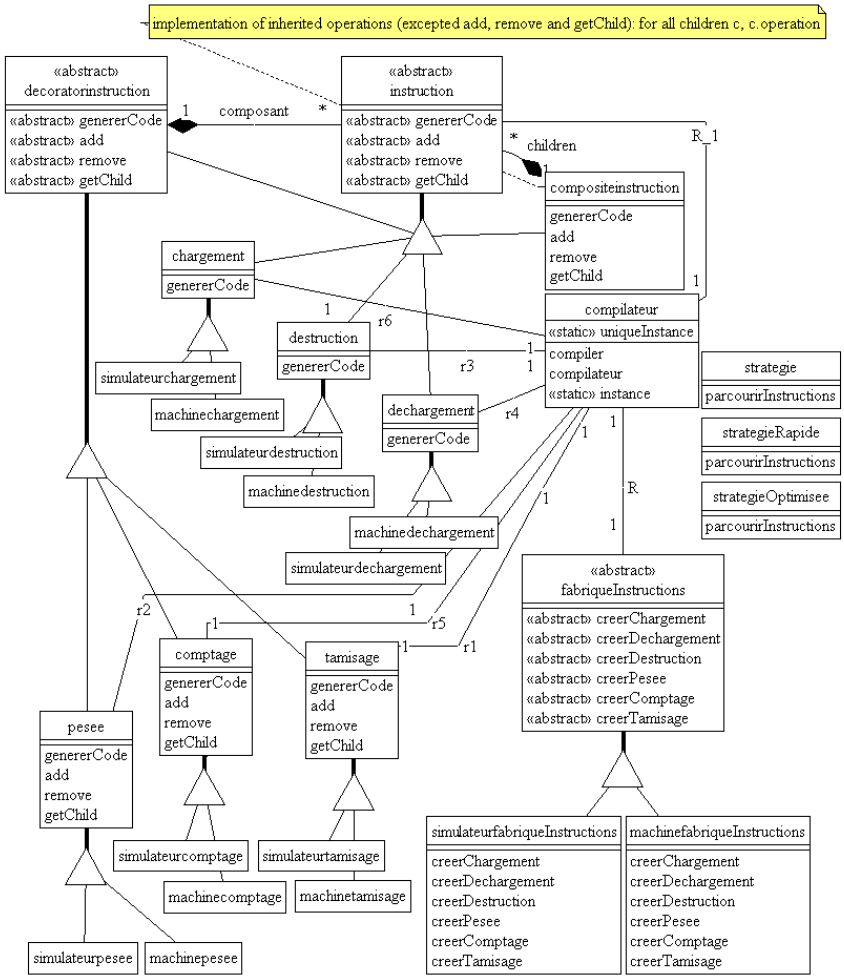

FIG. 8.7: Transformation du compilateur par Fabrique Abstraite

Le diagramme ainsi obtenu est illustré par la figure 8.8. Ce diagramme montre l'implémentation de STRATEGIE par STRATEGIERAPIDE et STRATEGIEOPTIMISEE et la composition de COMPILATEUR vers STRATEGIE, ce qui forme le DP Stratégie.

Le contexte de ce DP qu'est COMPILATEUR peut alors utiliser l'une ou l'autre des stratégies concrètes pour parcourir les instructions et ainsi générer le code destination selon que la compilation doit produire ce code rapidement – utilisant alors STRATEGIERAPIDE – ou qu'elle doit optimiser la séquence d'instructions – utilisant alors STRATEGIEOPTIMISEE.

Conclusion

Le diagramme obtenu à la figure 8.8 n'est pas le diagramme final sur lequel pourrait reposer une implémentation.

Pour obtenir un tel diagramme, nous devrions d'abord préciser nos transformations par des détails dont elles ne tiennent pas compte. Par exemple, devrions-nous propager les opérations genererCode des instructions vers leurs spécialisations que sont les produits concrets de la Fabrique Abstraite.

Ensuite, nous devrions énoncer clairement les choix d'implémentation que le diagramme de classes et les DP laissent ambigües et édicter les règles d'utilisations du diagramme qui ne sont pas directement induites par les DP. Par exemple, afin d'éviter qu'on ajoute une pesée à un bloc d'instructions, nous déciderions que les composants concrets du Décorateur ne comprennent pas compositeinstruction, c'est-à-dire que les instructions additionnelles – et décorateurs concrets – pesee, comptage et tamisage ne pourraient décorer cette classe. De même, nous devrions décider du destin des opérations add, remove et getChild dans ces décorateurs concrets : supprimées ou implémentées d'une manière particulière qui les rende inactives.

Finalement, nous devrions certainement compléter les concepts en reliant notamment la classe de stratégie optimisée strategieOptimisee au système expert décrit dans le problème.

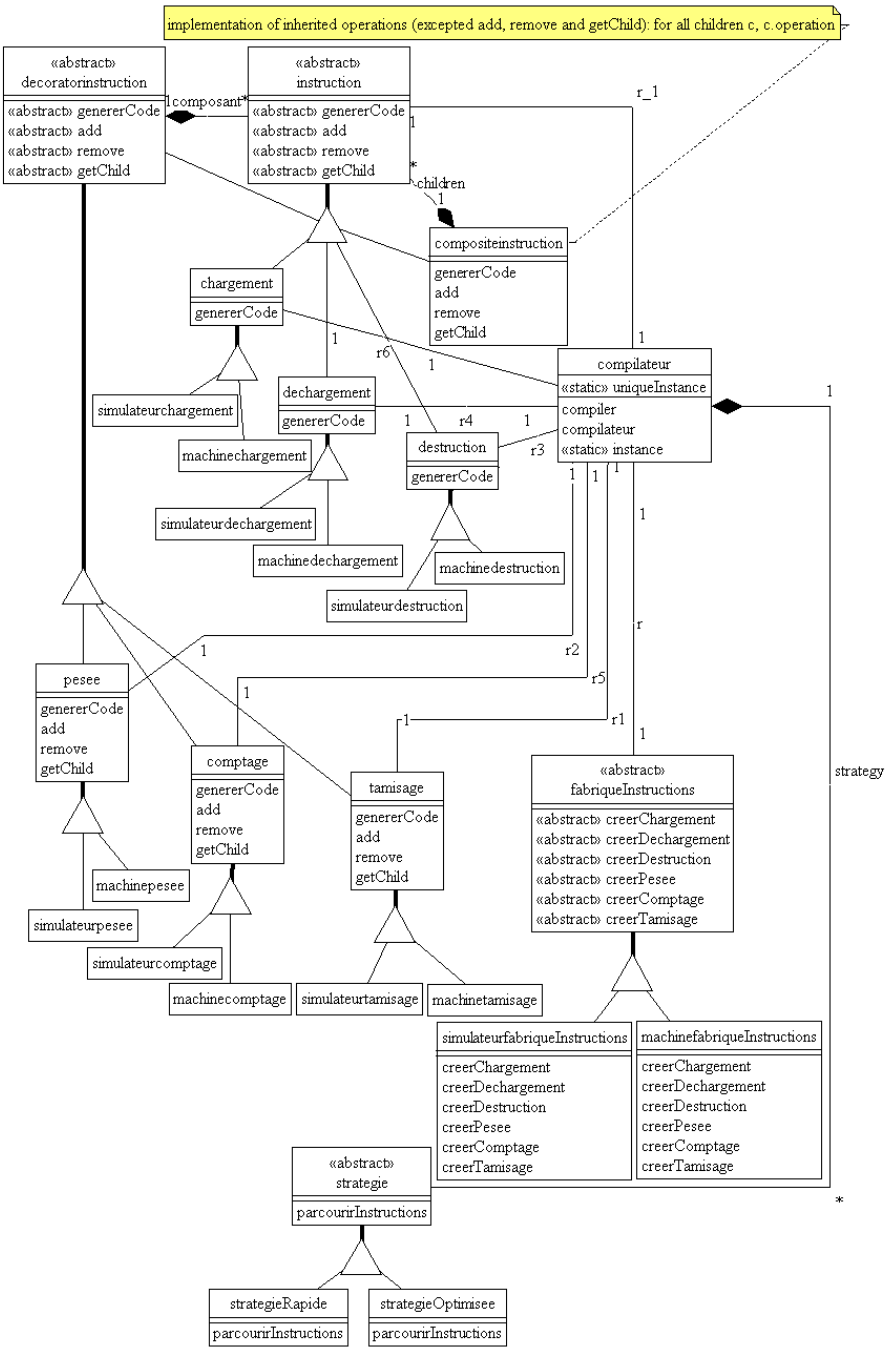

FIG. 8.8: Transformation du compilateur par Stratégie

Cinquième partie

Partie conclusive

Chapitre 9

Critique et suggestions

La description des transformations (chapitre 5) souligne combien les choix particuliers de conception de chacune d'entre-elles sont nombreux dans leur mise au point. Dans chacune de ces transformations, différentes possibilités nous apparaissent : ajouter ou retirer l'un ou l'autre rôle parmi ceux que nous avons établis ; donner plus ou moins de souplesse à chaque signature et accepter plus ou moins de paramètres, éventuellement sous d'autres formes que ceux requis ; contrôler plus ou moins de qualités du diagramme, sachant que chaque précondition est sujette à discution.

Bien plus, les transformations en elles-mêmes, définies par leurs tâches, peuvent toutes être remises en cause. Elles ne couvrent certainement pas tous les cas de figure et présentent forcément des imperfections. Par exemple, la transformation en COMPOSITE devrait créer les opérations héritées du COMPOSANT dans les FEUILLES, en omettant, peut-être, les opérations spécifiques à la classe COMPOSITE (ADD, REMOVE et GETCHILD).

Toutefois, ce sont les capacités intrinsèques à notre concept fondateur, l'enrichissement par approche transformationnelle[1], et les conséquences de nos choix technologiques, Prolog et DB-Main, que nous souhaitons mettre en avant et non les détails d'une implémentation qui se veut une "preuve de concept", par nature difficilement comparable au développement nanti de bien plus de ressources de travail propres à un cadre professionnel.

En particulier, le nombre des transformations que nous avons complétées est volontairement restreint : cinq sur les vingt-trois DP du livre *Design Patterns* [GoFDP]. Pourtant, nous pouvons déjà en tirer des enseignements très positifs.

9.1 Comparaison avec l'existant

Nous avons découvert dans le chapitre 3 qui étudie trois logiciels existant, un ensemble de particularités récurentes des outils CASE proposant l'enrichissement de diagrammes de classes à l'aide de DP. C'est avec ces logiciels, et plus

[1] Définie au chapitre 4.

précisément à la lumière de ces particularités, que nous souhaitons comparer notre implémentation. Nous ne nous attardons donc pas sur des considérations ergonomiques – dans notre cas bien dépendantes de DB-Main[2].

9.1.1 Inconvénients

Tout d'abord, notre capacité de méta-information est moins aboutie que dans les logiciels étudiés précédemment. Pour un utilisateur final n'accédant que via DB-Main et son *plug-in* à notre système de transformation, elle n'est présente que dans les commentaires accompagnant les écrans de "paramétrisation" et dans les notes que les transformations peuvent insérer dans le diagramme. Même si une seule transformation utilise cet ajout d'information par le biais des notes, en l'occurence COMPOSITE, ce système est extensible et permettrait facilement d'ajouter plus de méta-information aux diagrammes enrichis.

De plus, l'identification visuelle du DP dans le diagramme n'est pas favorisée au contraire de Borland Together grâce à son concept de "pattern" représenté par un ovale. Cette limitation est liée à la visualisation par DB-Main car le concept d'historique présent dans le noyau permet, en principe[3], une fonctionnalité similaire par l'exploitation de notre concept de *played_ transformation* définie en 5.2.1.

Enfin, contrairement aux logiciels étudiés, notre système ne propose pas de fonctionnalité de création de nouveau DP ouverte à l'utilisateur final. En effet, l'insertion d'une nouvelle transformation nécessiterait probablement un à deux jour(s) de travail pour un programmeur connaissant quelque peu Prolog. Pourtant, nous pouvons aisément relativiser cet inconvénient dans la mesure où il était annoncé avec le principe d'approche transformationnelle qui entend développer un ensemble de tâches particulières à chaque transformation. Dans ce contexte, le travail à fournir pour ajouter une transformation ne nous semble pas excessif au regard des bénéfices développés dans les lignes qui suivent.

9.1.2 Avantages

L'avantage le plus évident est sans doute le gain en souplesse : la versatilité de la "paramétrisation" permet de choisir un nombre d'éléments déterminé au moment de l'enrichissement et d'un type généralement plus large que ceux proposés dans les outils du chapitre 3.

Bien plus, l'adéquation avec l'esprit du DP est mieux respecté dans notre système qui n'impose pas simplement un diagramme de structure particulier.

Notons que ces deux premiers atouts sont bien visibles dans l'enrichissement par le DP FABRIQUE ABSTRAITE[4] et son concept de "famille" qui permet à

[2] A titre de rappel, DB-Main n'est pas fondé sur le concept UML et, naturellement, il ne propose pas toutes les capacités de visualisation attendues d'un outil UML.

[3] Par cette expression, nous voulons dire que l'historique présente dans le noyau contient en puissance toute l'information nécessaire à une telle fonctionnalité, mais que nous ne l'avons pas développée concrètement.

[4] Décrit en 7.5.

l'utilisateur de définir un ensemble de littéraux représentant les différentes familles de produits à fabriquer. La transformation pourra par ce biais contrôler l'existence d'autant de fabriques concrètes et de produits concrets (par produit abstrait) que de familles ainsi définies ; dans le cas contraire, il sera capable de les créer.

Un troisième avantage est l'implémentation d'un historique au niveau du *noyau*. Bien que cet historique ne soit pas particulièrement exploité dans notre système, il fournit les fondations d'un concept qui manquait aux trois logiciels étudiés : la possibilité de manipuler les transformations accomplies.

Finalement, notre système permet un bon contrôle – explicite – de la validité des diagrammes de base et enrichis. Cependant, ce contrôle de validité mériterait d'être développé plus encore, notamment en considérant le mode de fonctionnement des transformations comme des préconditions générales. Par exemple, nous pourrions refuser les classes abstraites dont les opérations ne seraient pas toutes abstraites dans la mesure où ces opérations ne seraient pas copiées dans d'hypotétiques classes spécialisées lors d'une transformation.

9.2 Conséquences des choix techniques

Les choix des techniques utilisées, que ce soit celui de Prolog pour la programmation du *noyau* ou de DB-Main pour l'interface homme-machine, impliquent des contraintes et des avantages.

Si nous avons déjà mis en avant les atouts de Prolog pour accomplir notre tâche, soulignons encore son aspect adapté à notre propos étant donné que programmer avec ce langage implique de définir clairement les concepts utilisés et leurs relations.

Bien sûr, Prolog n'était pas nécessaire et probablement pas optimal en terme de performance ou de concision du code, mais puisque son utilisation force à se concentrer sur le coeur du sujet traité plutôt que sur des moyens techniques sans rapport direct avec ce sujet, il provoque presque obligatoirement – par nature – une avancée dans la compréhension du système de connaissances dans le chef du programmeur, fait crucial dans l'optique d'une approche transformationnelle.

Par ailleurs, DB-Main constitue, après Prolog, l'autre choix technologique d'importance.

Son principal avantage, tel qu'exposé en 5.4.1, est la possibilité, via Voyager 2, d'interagir avec sa représentation interne des schémas.

Par contre, surtout à cause de sa meilleure adéquation aux schémas entité-association qu'aux diagrammes UML, DB-Main implique des limitations non propres au système en lui-même, c'est-à-dire des limitations que le *noyau* seul ne présente pas[5].

[5] Une manifestation facilement visible de ces limitations est la représentation graphique de DB-Main qui ne permet pas de lire la visibilité ou les paramètres d'une opération (*processing unit*) sans manipulation spécifique.

A titre d'exemple, citons la perte des notes lors de l'export du diagramme de DB-Main vers le noyau (équivalent à la transition EXPORT définie en 5.4.2), bien que cet inconvénient soit très probablement soluble par l'amélioration du composant DPT_Diagram_Export[6] de notre système.

En définitive, épinglons l'obstacle principal induit par notre utilisation de DB-Main : la perte des notions d'historique. En effet, les enrichissements successifs[7] supposent le passage du diagramme au *noyau* lors de chaque transformation comme s'il était un diagramme neuf. En d'autres termes, à chaque transformation, toute la connaissance du *noyau* est réinitialisée et tout ce qui n'est pas connu de DB-Main est perdu.

9.3 Conclusion de la critique

En conclusion, les inconvénients de notre système de transformation sont principalement ceux d'une preuve de concept qui nécessiterait plus d'investissement, tel que, par exemple, la mise au point d'une interface de visualisation sur-mesure.

Sa qualité principale est induite par son concept d'approche transformationnelle : comme attendu, nos transformations se définissent par des tâches spécifiques à chaque DP ce que nous considérons comme l'avancée principale par rapport aux outils CASE précédemment critiqués dans la mesure où c'est cette qualité qui permet le gain en souplesse et surtout la meilleure adéquation aux concepts du DP, y compris par la vérification de préconditions *ad hoc*.

Bien sûr, l'avantage induit son défaut et la spécificité s'opposant à la généricité, la création d'une nouvelle transformation demande plus d'effort à l'utilisateur que dans les logiciels étudiés dans l'état de l'art.

9.4 Suggestions

9.4.1 Améliorations

Au titre de prototype, notre système mériterait plusieurs améliorations. A ce stade de notre recherche, nous pouvons suggérer les suivantes.

Premièrement, l'historique devrait être propagé à l'interface utilisateur, ce qui aurait deux avantages : d'abord, pouvoir le visualiser et l'utiliser via cette interface pour, par exemple, implémenter une fonctionnalité d'annulation de transformations accomplies ; secondement, en garder la connaissance lors de transformations successives et ainsi pouvoir en bénéficier lors de la vérification des préconditions.

Ensuite, d'autres transformations à l'aide d'autres DP devraient être programmées. Par contre, la mise en place d'un système de création de transformation par l'utilisateur final nous semble difficile ; une telle fonctionnalité impliquerait

[6] Défini en 5.4.1.

[7] Tels que démontrés au chapitre 8

la mise en place d'une procédure de méta-programmation qui, pour être utile, devrait être plus aisée d'utilisation que la programmation en Prolog elle-même.

De surcroît, la méta-information fournie par notre système devrait être améliorée. Soulignons différents apports possibles qui nous paraissent intéressants : la propagation de la documentation telle que décrite en 5.2.1 vers l'interface utilisateur et la programmation des fonctionnalités permettant d'utiliser cette documentation ainsi que l'intégration dans cette documentation de la description des DP et des explications concernant chacun de ses rôles au sein de ce DP et leurs liens avec les paramètres de la transformation.

Enfin, nos signatures actuelles ne permettent pas de choisir le nom des éléments créés mais non identifiés de la transformation. Par exemple, lorsque la transformation en Fabrique Abstraite crée les produits concrets non identifiés[8], elle leur donne le nom consistant en la concaténation du nom de famille adéquat (un paramètre de type littéral) et du nom du produit abstrait dont le produit concret hérite[9]. Un utilisateur pourrait souhaiter choisir ces noms d'éléments du diagramme que le système va engendrer. L'amélioration consisterait donc à proposer des signatures alternatives permettant d'entrer les noms par défaut des éléments sensés être générés.

9.4.2 Alternatives

Des alternatives aux choix de conceptions que nous avons posés sont bien sûr envisageables . En voici deux qui nous semblent pertinentes.

D'abord, la visualisation des diagrammes aurait pu être confiée à bien d'autres outils que DB-Main à condition de disposer d'un moyen de description standard des diagrammes de classes par opposition aux fichiers ISL spécifiques à DB-Main. Le standard XMI[10] (XML Meta Interchange) aurait pu jouer ce rôle, mais la traduction de notre représentation du diagramme sous la forme de fichiers conformes à ce standard et l'analyse des diagrammes spécifiés ainsi afin d'être intégrés dans notre *noyau* auraient nécessité un travail que nous avons estimé peu pertinent pour notre propos.

Néamoins, eut-il encore fallut y intégrer ensuite le concept d'historique et plus encore trouver une solution pour diriger les transformations à partir de la nouvelle interface utilisateur, c'est-à-dire pour remplacer les fonctionnalités programmées à l'aide de Voyager 2.

---

[8] Nous entendons par là, non passés comme paramètres.

[9] Ce nom, comme tous les noms créés par le système, est éventuellement renommé en y accolant un chiffre derrière de manière à ce que la classe créée ait un nom unique dans le diagramme.

[10] Défini par l'OMG comme "Un cadre d'intégration XML propulsé par modèle pour définir, échanger, manipuler et intégrer des données et object XML. Les standards basés sur XMI sont utilisés par des outils d'intégration, des référentiels (*repository*), des applications et entrepôts de données (*data warehouse*). XMI fournit des règles par lesquelles un schéma peut être généré pour tout méta-modèle MOF transmissible par XMI." [OMG], *in* `http://www.omg.org/technology/documents/modeling_spec_catalog.htm#XMI`(consulté le 27 mai 2006) ("propulsé par modèle" est une traduction de *model driven*.)
Pour comprendre cette définition, il convient de préciser qu'un diagramme de classes UML peut, en particulier, être modélisé par un méta-modèle MOF. De nombreux outils CASE centrés sur l'UML utilisent des fichiers XMI comme standard d'échange de données.

Ensuite, l'approche transformationnelle aurait pu être implémentée alternativement par un langage de transformation tel que décrit dans le contexte du MDA[11] mis au point par l'Object Management Group mieux connu comme OMG.

En particulier, un langage répondant au nom de ATL[12], répond aux spécifications de MDA pour un langage de transformation. "Le langage de transformation Atlas (ATL) est un langage hybride (un mélange de constructions déclaratives et impératives) conçu pour exprimer des modèles de transformations comme requis par l'approche MDA pour répondre au QVT RFP[13] publié par OMG"[14].

Programmer en ATL consiste à spécifier des règles[15] de transformation définies d'un méta-modèle source vers un méta-modèle cible. Ces règles permettent de créer une machine de transformation capable de transformer les modèles dont le méta-modèle source est un méta-modèle en un modèle dont le méta-modèle cible est un méta-modèle.

Ce langage en partie déclaratif nous aurait peut-être permis d'arriver à des résultats similaires à ceux effectivement produits. Néanmoins, le fait que les modèles et méta-modèles utilisés et produits par ATL lors de la programmation et surtout lors des transformations sont exprimés en XMI, nous permet de supposer que la visualisation des diagrammes aurait alors pu être prise en charge par une variété d'outils CASE avec un minimum de programmation spécifique à cet égard.

---

[11]MDA (MDA `http://www.omg.org/mda/`) est l'acronyme de *Model-Driven Architecture* (qu'on pourrait traduire par "architecture propulsé par modèle") mis au point par l'Object Management Group mieux connu comme OMG. MDA est développé par l'OMG. "MDA est une approche pour utiliser des modèles dans le développement logiciel." [MDAG, p. 2-1]
"Les trois objectifs principaux de MDA sont la portabilité, l'interopérabilité et la réutilisabilité"[MDAG, p 2-2]. Le concept de MDA est de décrire un système plannifié (en ce concentrant sur les logiciels dans ce système) à l'aide de modèles. Ces modèles sont au coeur de MDA qui en proposera certains types, comment les utiliser et les relier. MDA va utiliser trois points-de-vue (*Viewpoint*) sur un système : le point de vue indépendant de l'ordinateur, le point de vue indépendant de la plateforme et le point de vue spécifique à la plateforme. Ces trois points-de-vue conduisent à trois modèles : CIM, le modèle indépendant de l'ordinateur (*Computer Independant Model*) ; PIM, le modèle indépendant de la plateforme (*Platform Independant Model*) ; PSM, le modèle spécifique à la plateforme (*Platform Specific Model*). MDA manipulant de multiples points-de-vue et de multiples modèles d'un sytème va naturellement s'intéresser à une automatisation du passage d'un modèle à un autre dans le même système : c'est le concept de transformation de modèle.
"Une *transformation de modèle* est le processus de conversion d'un modèle vers un autre modèle du même système."[MDAG, p 2-7]

[12]ATL est l'acronyme de *Atlas Transformation Language*. Une présentation du langage ATL peut se découvrir via l'URL suivant : `http://www.sciences.univ-nantes.fr/lina/atl/atlProject/presentation/` (consulté le 28 mai 2006).

[13]QVT RFP est l'acronyme de *Query View Transformation Request For Proposal*, la demande de l'OMG de proposition d'implémentation d'un outil de transformation conforme à leurs spécifications qui peut se lire via cet URL : `http://www.omg.org/docs/ad/02-04-10.pdf` (consulté le 28 mai 2006).

[14]The Atlas Project, Université de Nantes, `http://www.sciences.univ-nantes.fr/lina/atl/atlProject/presentation/` (consulté le 28 mai 2006).

[15]Les requêtes ATL contiennent des expressions en langage OCL. OCL est l'acronyme de *Object Constraint Language*, un langage défini par l'OMG dans l'UML (lui-même inclus dans le cadre MDA). La définition de l'OCL peut se trouver en `http://www.omg.org/docs/ptc/05-06-06.pdf` (consulté le 28 mai 2006, mais ces spécifications ne sont pas anoncées définitives).

Le temps disponible ne nous a pas permis de nous pencher plus profondément sur ce langage qui reste, nous le croyons, une alternative intéressante à explorer par ailleurs.

Chapitre 10

Conclusion générale

Notre recherche s'est attachée à aborder l'enrichissement de diagrammes de classes à l'aide de design patterns sous l'angle des transformations de modèles. Peu de logiciels exploitent cette notion à ce jour, c'est pourquoi nous avons mis au point un système de transformation au titre de "preuve de concept" dans le but de pouvoir critiquer l'approche transformationnelle à la lumière d'une implémentation concrète.

Dans les chapitres précédents explicitant notre solution et nos conclusions, nous avons d'abord situé la question de départ dans son contexte par la définition des design patterns et la mise en exergue de leurs principaux intérêts[1] : nous avons vu comment les design patterns pouvaient aider à la conception d'une programmation orientée objet.

Nous avons, par la suite, analysé trois outils CASE et plus précisément, leurs qualités d'enrichissement de diagrammes par design patterns. Cette critique a permis de dégager des limites récurentes dont un manque de souplesse dans la "paramétrisation" de l'insertion du design pattern et une inadéquation à l'esprit spécifique du design pattern. Ces outils ajoutent ou identifient sans modifier des éléments au diagramme selon une méthode générique propre à chaque outil[2].

En conséquence, nous avons pu définir l'approche transformationnelle par opposition à cette approche générique en nous fixant l'objectif de définir une transformation spécifique à chaque design pattern. Les objectifs ainsi balisés, nous avons mis en évidence les raisons pour lesquelles Prolog nous semblait adéquat à leur réalisation[3].

Puis, nous avons décrit notre solution pour développer un système de transformation. Nous y avons modélisé notre système de connaissances avant de décrire son implémentation physique et son intégration dans DB-Main rendue possible grâce au langage Voyager 2[4].

De plus, nous avons complété la description de notre proposition par la liste des transformations effectivement implémentées et de leurs caractéristiques[5].

[1] Voir chapitre 2.
[2] Voir chapitre 3.
[3] Voir chapitre 4.
[4] Voir chapitre 5.
[5] Voir chapitre 7.

Cette description s'est suivie naturellement d'une étude de cas afin de montrer un exemple concret d'utilisation de notre système. Nous avons ainsi pu obtenir un diagramme de classes solutionnant un problème exemplatif à l'aide de cinq design patterns[6].

Le temps était alors venu de critiquer notre solution à la lumière des résultats obtenus. La comparaison avec l'existant de notre système de transformation a mis en évidence des inconvénients que nous pouvons considérer comme mineurs et qui sont propres à une "preuve de concept" ;un autre type d'inconvénient consiste en une perte d'information entre le *noyau* du système et son interface visuelle, DB-Main, notamment de l'historique des transformations.

De plus, la perte de généricité, engendrée par notre approche transformationnelle spécifique, nous empèche de développer facilement une procédure d'ajout par l'utilisateur de nouveaux design patterns dans le catalogue des transformations tels que le proposent les trois autres outils étudiés.

En revanche, le gain en souplesse et en adéquation à l'intention du design pattern nous est clairement apparu comme l'avantage majeur de notre approche tandis que la notion d'historique et les vérifications de validité des diagrammes et de précondition des transformations plaidaient également pour notre système d'enrichissement.

En outre, nous avons proposé des pistes d'amélioration de notre prototype par la propagation de l'historique des tranformations vers son interface visuelle, l'ajout de nouvelles transformations à l'aide d'autres design patterns, l'amélioration de la méta-information ou encore la possibilité de nommer les nouveaux éléments du diagramme par l'assouplissement des signatures.

Pour clore nos investigations, nous avons évoqué des alternatives à nos choix technologiques en discutant d'abord brièvement de l'échange des informations définissant le diagramme sous forme d'un format standard, en l'occurence XMI, par opposition au développement spécifique à DB-Main des fichiers ISL ; nous avons aussi présenté succinctement ATL, un langage spécifiquement orienté vers les transformations de modèles, comme une alternative au choix de Prolog dans notre développement[7].

[6]Voir chapitre 8.
[7]Voir, pour le développement de ces cinq précédents paragraphes, chapitre 9.

Sixième partie

Annexes

Annexe A

Exemples choisis de la programmation Prolog

Pour des raisons de place disponible, nous ne pouvons pas fournir ici tout le code Prolog de notre prototype qui compte plusieurs milliers de lignes[1]. Néanmoins, nous avons choisis quelques passages que nous croyons significatifs.

A.1 DPT

DPT est le programme central du noyau, tel que défini au chapitre 5.

A.1.1 Prédicats dynamiques

Les prédicats dynamiques sont les prédicats succeptibles d'être manipulés dynamiquement, c'est-à-dire par méta-programmation.

```
:-dynamic
    member_visibility/2,
    member_name/2,
    member_stereotype/2,
    attribute/1,
    operation/1,
    operation_parameters/2,
    has/2,
    class/1,
    class_name/2,
    class_stereotype/2,
    class_properties/2,
    link/1,
    composition/1,
```

[1]Pour les mêmes raisons, nous ne fournissons pas le code Voyager 2 du *plug-in* DPT_Diagram_Export qui sert à la programmation de l'interface utilisateur.

```
    generalization/1,
    association_name/2,
    left_end/2,
    right_end/2,
    note/2,
    check_result/2,
    increment/1,
    last_played_transfo_num/1,
    played_transfo/2,
    transformed/3.
```

A.1.2 Contraintes d'intégrité

Les contraintes d'intégrité vérifient la validité du diagrammes. Ces contraintes vérifient que les classes aient des noms différents, que les méthodes d'une même classe aient des signatures différentes et que les attributs d'une même classe aient des noms différents.

```
check_constraints(Res) :-
    check_result_ok,
    writef("\nCheck constraints : ",[]),
    all_class_distinct([]),
    check_result(Res,L),
    write_res(Res,L).
check_constraints(Res) :-
    check_result(Res,L),
    write_res(Res,L).
check_result_ok :-
    remove_check_result,
    assert(check_result(ok,[])).
remove_check_result :-
    check_result(A,B),
    !,
    retract(check_result(A,B)).
remove_check_result.
all_class_distinct(L) :-
    class(X),
    not(member(X,L)),
    !,
    verif_class_name(X,L),
    all_method_distinct(X,[]),
    all_attribute_distinct(X,[]),
    L1=[X|L],
    !,
    all_class_distinct(L1).
```

```
all_class_distinct(_).
verif_class_name(C,L) :-
    class_name(C,Name),
    member(X,L),
    class_name(X,Name),
    !,
    res_vcn(Name),
    fail.
verif_class_name(_,_).
res_vcn(N) :-
    remove_check_result,
    assert(check_result(class_same_name,[N])).
%%vérifier signature et pas nom car overload permis
all_method_distinct(C,L) :-
    operation(X),
    has(C,X),
    not(member(X,L)),
    !,
    verif_ope_signature(C,X,L),
    verif_param(C,X),
    L1=[X|L],
    !,
    all_method_distinct(C,L1).
all_method_distinct(_,_).
verif_ope_signature(C,O,L) :-
    member_name(O,Name),
    member(X,L),
    member_name(X,Name),
    same_types(X,O),
    !,
    res_vos(C,Name),
    fail.
verif_ope_signature(_,_,_).
same_types(O1,O2) :-
    operation_parameters(O1,L1),
    !,
    operation_parameters(O2,L2),
    st_list(L1,L2).
same_types(_,O2) :-
    not(operation_parameters(O2,_)).
st_list([[_,Type1]|T1],[[_,Type2]|T2]) :-
    Type1 = Type2,
    st_list(T1,T2).
st_list([[_]|T1],[[_]|T2]) :-
    st_list(T1,T2).
```

```
st_list([],[]).
res_vos(C,OpeName) :-
    remove_check_result,
    class_name(C,ClassName),
    assert(check_result(ope_same_signature,[ClassName,OpeName])).
verif_param(C,O) :-
    operation_parameters(O,PList),
    !,
    res_vp_pl(C,O),
    parameters_list(PList),
    res_vp_apd(C,O),
    all_param_distinct(PList),
    check_result_ok.
verif_param(_,_).
res_vp_pl(C,O) :-
    remove_check_result,
    class_name(C,ClassName),
    member_name(O,OpeName),
    assert(check_result(wrong_param,[ClassName,OpeName])).
res_vp_apd(C,O) :-
    remove_check_result,
    class_name(C,ClassName),
    member_name(O,OpeName),
    assert(check_result(param_same_name,[ClassName,OpeName])).
%tous les noms d'une liste de paramètres doivent être distincts
all_param_distinct([[N,_]|T]) :-
    testParamName(T,N),
    !,
    all_param_distinct(T).
all_param_distinct([[N]|T]) :-
    testParamName(T,N),
    !,
    all_param_distinct(T).
all_param_distinct([]).
%%all_attribute_distinct vérifie que
%les attributs d'une même classe aient un nom distinct
all_attribute_distinct(C,L) :-
    attribute(X),
    has(C,X),
    not(member(X,L)),
    !,
    verif_attrib_name(C,X,L),
    L1=[X|L],
    !,
    all_attribute_distinct(C,L1).
```

```
all_attribute_distinct(_,_).
verif_attrib_name(C,A,L) :-
    member_name(A,Name),
    member(X,L),
    member_name(X,Name),
    !,
    res_van(C,Name),
    fail.
verif_attrib_name(_,_,_).
res_van(C,AttribName) :-
    remove_check_result,
    class_name(C,ClassName),
    assert(check_result(attrib_same_name,[ClassName,AttribName])).
```

A.1.3 Transformations

Voici le code des transfomations ; nous en avons retiré les trops longues procédures spécifiques, préconditions, enregistrement des transformations et documentation pour nous concentrer sur les tâches de la transformation.

A.1.3.1 Singleton

```
transfo(singleton,[arg_class_name(N)]) :-
    check_constraints(ConsRes),
    ConsRes = ok,
    class(X),
    class_name(X,N),
    writef("\nCLASS TO TRANSFORM\n",[]),
    visualise(N),
    !,
    singletonize(X),
    writef("singletonize OK\n",[]),
    writef("\nCLASS TRANSFORMED\n",[]),
    visualise(N),
    check_constraints(_).
singletonize(C) :-
    precondition(singleton,[C]),
    !,
    add_played_transfo(singleton),
    constructeur(C,Cons),
    writef("\n",[]),
    remove_visibility(Cons),
    assert(member_visibility(Cons,protected)),
    add_transformed(Cons,constructeur),
    assert_attribute(C,default_name(uniqueInstance),
    visib(private),stereo(static),NewA),
    add_transformed(NewA,uniqueInstance),
```

```
    assert_operation(C,default_name(instance),
    visib(public),stereo(static),NewO),
    add_transformed(NewO,instance),
    add_transformed(C,singleton).
singletonize(_) :-
    transfo_fail.
```

A.1.3.2 Composite

```
transfo(composite,[composite(Composite),feuilles(L),
client(Client)]) :-
    transfo(composite,[composant(ignored),
    composite(Composite),feuilles(L),client(Client)]).
transfo(composite,[composant(Composant),feuilles(L),
client(Client)]) :-
    transfo(composite,[composant(Composant),
    composite(ignored),feuilles(L),client(Client)]).
transfo(composite,[composite(Composite),feuilles(L)]) :-

    transfo(composite,[composant(ignored),
    composite(Composite),feuilles(L),client(ignored)]).
transfo(composite,[composant(Composant),feuilles(L)]) :-

    transfo(composite,[composant(Composant),
    composite(ignored),feuilles(L),client(ignored)]).
transfo(composite,[composant(Composant),
composite(Composite),feuilles(L)]) :-
    transfo(composite,[composant(Composant),
    composite(Composite),feuilles(L),client(ignored)]).
transfo(composite,[composant(Composant),
composite(Composite),feuilles(L),client(Client)]) :-
    check_constraints(ConsRes),
    ConsRes = ok,
    prepare_compositize(Composant,Composite,L,
    Client,Co,Ci,F,Cl),
    !,
    compositize(Co,Ci,F,Cl),
    writef("compositize OK\n",[]),
    writef("\nCLASSES TRANSFORMED\n",[]),
    visualise(all),
    check_constraints(_).
transfo(composite,_) :-

    writef("\n'composite' called with bad arguments\n",[]),
    !,
    fail.
```

```
prepare_compositize(Composant,ignored,Feuilles,ignored,Co,
Ci,F,Cl) :-
    class(Co),
    class_name(Co,Composant),
    Ci = ignored,
    term_in_names(Feuilles,FL,feuille),
    class_list(FL,F),
    Cl = ignored,
    writef("\nCLASSES TO TRANSFORM\n",[]),
    visualise(Composant),
    visualise(FL).
prepare_compositize(ignored,Composite,Feuilles,ignored,Co,
Ci,F,Cl) :-
    Co = ignored,
    class(Ci),
    class_name(Ci,Composite),
    term_in_names(Feuilles,FL,feuille),
    class_list(FL,F),
    Cl = ignored,
    writef("\nCLASSES TO TRANSFORM\n",[]),
    visualise(Composite),
    visualise(FL).
prepare_compositize(ignored,Composite,Feuilles,Client,Co,
Ci,F,Cl) :-
    Co = ignored,
    class(Ci),
    class_name(Ci,Composite),
    term_in_names(Feuilles,FL,feuille),
    class_list(FL,F),
    class(Cl),
    class_name(Cl,Client),
    writef("\nCLASSES TO TRANSFORM\n",[]),
    visualise(Composite),
    visualise(FL),
    visualise(Client).
prepare_compositize(Composant,ignored,Feuilles,Client,Co,
Ci,F,Cl) :-
    class(Co),
    class_name(Co,Composant),
    Ci = ignored,
    term_in_names(Feuilles,FL,feuille),
    class_list(FL,F),
    class(Cl),
    class_name(Cl,Client),
    writef("\nCLASSES TO TRANSFORM\n",[]),
    visualise(Composant),
    visualise(FL),
    visualise(Client).
```

```
prepare_compositize(Composant,Composite,Feuilles,ignored,Co,
Ci,F,Cl) :-
    class(Co),
    class_name(Co,Composant),
    class(Ci),
    class_name(Ci,Composite),
    term_in_names(Feuilles,FL,feuille),
    class_list(FL,F),
    Cl = ignored,
    writef("\nCLASSES TO TRANSFORM\n",[]),
    visualise(Composant),
    visualise(Composite),
    visualise(FL).
prepare_compositize(Composant,Composite,Feuilles,Client,Co,
Ci,F,Cl) :-
    class(Co),
    class_name(Co,Composant),
    class(Ci),
    class_name(Ci,Composite),
    term_in_names(Feuilles,FL,feuille),
    class_list(FL,F),
    class(Cl),
    class_name(Cl,Client),
    writef("\nCLASSES TO TRANSFORM\n",[]),
    visualise(Composant),
    visualise(Composite),
    visualise(FL),
    visualise(Client).
compositize(Composant,Composite,FList,Client) :-

    precondition(composite,[Composant,Composite,FList,Client]),
    !,
    add_played_transfo(composite),
    %1. Composant devient abstract et création des opérations
    %add, remove et getChild. si 'ignored' composant est une
    %copie de composite
    create_composant(Composant,Composite,Ca),
    add_transformed(Ca,composant),
    %2. Ajout dans la classe composite des opération add,
    %remove et getChild. si 'ignored' composite est une copie
    %de composant. ajout d'une note pour le composite.
    create_composite(Composite,Ca,Ci),
    assert_note(Ci,'implementation of inherited operations

        (excepted add, remove and getChild) : for all
        children c, c.operation'),
    add_transformed(Ci,composite),
```

```
    %3. Création des généralisations de composant vers les
    %feuilles et composite.
    create_generalization(Ca,Ci,_),
    generalize_list(Ca,FList,[],_),
    add_transformed_list(FList,feuille),
    %4. Création de la composition 'children' de composite vers
    %composant.
    create_composition(Ci,Ca,Composition),
    assert(association_name(Composition,children)),
    %5. Création du link entre client et composant.
    %Si 'ignored', ne fait rien.
    link_client_composant(Client,Ca).
compositize(_,_,_) :-
    transfo_fail.
```

A.1.3.3 Stratégie

```
transfo(strategy,[contexte(Context),strategy(Strategy),
concreteStrategies(L)]) :-
    check_constraints(ConsRes),
    ConsRes = ok,
    prepare_strategize(Context,Strategy,L,Co,S,CSL),
    !,
    strategize(Co,S,CSL),
    writef("strategize OK\n",[]),
    writef("\nCLASSES TRANSFORMED\n",[]),
    visualise(all),
    check_constraints(_).
transfo(strategy,_) :-

    writef("\n'strategy' called with BAD ARGUMENTS\n",[]),
    fail.
prepare_strategize(Context,Strategy,L,Co,S,CSL) :-
    class(Co),
    class_name(Co,Context),
    class(S),
    class_name(S,Strategy),
    term_in_names(L,TempL,concreteStrategy),
    class_list(TempL,CSL),
    writef("\nCLASSES TO TRANSFORM\n",[]),
    visualise(Context),
    visualise(Strategy),
    visualise(TempL).
strategize(Context,Strategy,ConcreteList) :-

    precondition(strategy,[Context,Strategy,ConcreteList]),
    !,
```

```
    add_played_transfo(strategy),
    %1. Strategy devient abstract
    assert_class_stereotype(Strategy,abstract),
    add_transformed(Strategy,strategy),
    %2. context
    add_transformed(Context,context),
    %3. ConcreteList -> generalisation de Strategy
    %vers chaque ConcreteStrategy class
    generalize_list(Strategy,ConcreteList,[],_),
    add_transformed_list(ConcreteList,concreteStrategy),
    %4. Création de la composition 'strategy'
    %de context vers strategy.
    create_composition(Context,Strategy,Composition),
    assert(association_name(Composition,strategy)).

compositize(_,_,_) :-
    transfo_fail.
```

A.1.3.4 Décorateur

Nous n'insérons pas la transformation en décorateur déjà décrite dans le chapitre 6.

A.1.3.5 Fabrique Abstraite

```
transfo(abstractFactory,[fabriqueAbstraite(FabriqueAbstraite),
familles(Familles),client(Client),produits(Produits)]) :-



    transfo(abstractFactory,[fabriqueAbstraite(FabriqueAbstraite),
    familles(Familles),client(Client),produits(Produits),
    fabriquesConcretes(ignored)]).

transfo(abstractFactory,[fabriqueAbstraite(FabriqueAbstraite),
familles(Familles),client(Client),produits(Produits),
fabriquesConcretes(FabriquesConcretes)]) :-

    check_constraints(ConsRes),
    ConsRes = ok,
    prepare_abstractFactorize(FabriqueAbstraite,Familles,Client,
    Produits,FabriquesConcretes,FA,F,Cl,P,FC),
    !,
    abstractFactorize(FA,F,Cl,P,FC),
    writef("abstractFactorize OK\n",[]),
    writef("\nCLASSES TRANSFORMED\n",[]),
    visualise(all),
    check_constraints(_).

transfo(abstractFactory,_) :-
```

```
    writef("\n'abstractFactory' called with BAD ARGUMENTS\n",[]),
    fail.
prepare_abstractFactorize(FabriqueAbstraite,Familles,
Client,Produits,ignored,FA,F,Cl,P,FC) :-
    class(FA),
    class_name(FA,FabriqueAbstraite),
    term_in_names(Familles,F,famille),
    class(Cl),
    class_name(Cl,Client),
    class_product(Produits,P),
    FC = [],
    writef("\nCLASSES TO TRANSFORM\n",[]),
    visualise(FabriqueAbstraite),
    visualise(Client),
    product_visualisation(Produits).
prepare_abstractFactorize(FabriqueAbstraite,Familles,
Client,Produits,FabriquesConcretes,FA,F,Cl,P,FC) :-
    class(FA),
    class_name(FA,FabriqueAbstraite),
    term_in_names(Familles,F,famille),
    class(Cl),
    class_name(Cl,Client),
    class_product(Produits,P),
    term_in_names(FabriquesConcretes,FCL,fabriqueConcrete),
    class_list(FCL,FC),
    writef("\nCLASSES TO TRANSFORM\n",[]),
    visualise(FabriqueAbstraite),
    visualise(Client),
    product_visualisation(Produits),
    visualise(FCL). %fabriques concrètes list
abstractFactorize(FA,Familles,Client,Produits,FC) :-

    precondition(abstractFactory,[FA,Familles,Client,

       Produits,FC]),
    !,
    add_played_transfo(abstractFactory),
    %1. créer une association « link » de client vers la FA
    %et vers chaque produit abstrait.
    create_link(Client,FA,_),
    abstract_products(Produits,AbstractProducts),
    link_list(Client,AbstractProducts,_),
    add_transformed(FA,fabriqueAbstraite),
    add_transformed(Client,client),
    %2. assurer que chaque famille ait sa fabrique concrète.
    create_concreteFactory(FA,Familles,FC,
```

```
    NewFabriquesConcretes),
    %NewFabriquesConcretes est dans l'ordre
    %inverse des familles
    add_transformed_list(NewFabriquesConcretes,
    fabriqueConcrete),
    %3. si elles nexistent déjà, créer
    %les associations « généralization »
    %de FA vers chaque fabrique concrète.
    generalize_list(FA,NewFabriquesConcretes,[],_),
    %4. Pour chaque produit, vérifier que tous
    %les produits concrets existent.
    create_concreteProducts(Familles,Produits,NewProduits),
    add_transformed_list(AbstractProducts,produitAbstrait),
    add_transformed_Products(NewProduits,produitConcret),
    %5. si elles nexistent déjà, créer les associations
    %de généralisation de chaque produit abstrait
    %vers ses produits concrets.
    generalize_products(NewProduits).
abstractFactorize(_,_,_,_,_) :-
    transfo_fail.
```

A.2 TMM

TMM est le programme Prolog qui permet la publication de la liste des transformations et de leurs signatures.

A.2.1 Procédures de publication

```
:-dynamic
    output_file/1,
    input_file/1.
output_file('C :/exchange/output.xml').
input_file('C :/exchange/input').
set_output_file(NewOutput) :-
    retract(output_file(_)),
    assert(output_file(NewOutput)).
set_input_file(NewInput) :-
    retract(input_file(_)),
    assert(input_file(NewInput)).
%la liste de toutes les transformations
%transfo_list
transfo_list :-
    output_file(Default),
    transfo_list(Default).
transfo_list(Out) :-
```

```
    writef("printing transfo list in %d file.\n",[Out]),
    tell(Out),
    writef("<transfoList>\n"),
    transfo_list_print([]),
    writef("</transfoList>"),
    told,
    writef("printing done.\n",[]).
transfo_list_print(L) :-
    signature(T,_),
    not(member(T,L)),
    !,
    L1 = [T|L],
    writef("<transfo>%d</transfo>\n",[T]),
    transfo_list_print(L1).
transfo_list_print(_).
%model(+Transfo)
model(Transfo) :-
    signature(Transfo,parameters(PL)),
    !,
    output_file(Out),
    writef("printing signature of the transformation '%d'
        in %d file.\n",[Transfo,Out]),
    tell(Out),
    writef("<signature>\n"),
    writef("<transfoName>%d</transfoName>\n",[Transfo]),
    writef("<parameters>\n"),
    write_model_parameters(PL),
    writef("</parameters>\n"),
    writef("</signature>"),
    told,
    writef("printing done.\n",[]).
write_model_parameters([parameter(ParameterName,IsFacultative,
ParameterType)|T]) :-
    writef("<parameter>\n"),
    writef("<parameterName>%d</parameterName><facultative>%d


        </facultative>\n",[ParameterName,IsFacultative]),
    write_model_parameter_type(ParameterType),
    writef("</parameter>\n"),
    write_model_parameters(T).
write_model_parameters([parameter(ParameterName,
ParameterType)|T]) :-
    writef("<parameter>\n"),
    writef("<parameterName>%d</parameterName>\n",
```

```
        [ParameterName]),
    write_model_parameter_type(ParameterType),
    writef("</parameter>\n"),
    write_model_parameters(T).
write_model_parameters([]).
write_model_parameter_type([H|T]) :-
    wmpt(H),
    write_model_parameter_type(T).
write_model_parameter_type([]).
wmpt(list(Param,MinParam,MaxParam)) :-
    writef("<listParameterType>\n"),
    writef("<min>%d</min><max>%d</max>\n",
        [MinParam,MaxParam]),
    write_model_parameters([Param]),
    writef("</listParameterType>\n").
%pas de maxparam si on considère l'"infini"
wmpt(list(Param,MinParam)) :-
    writef("<listParameterType>\n"),
    writef("<min>%d</min>\n",[MinParam]),
    write_model_parameters([Param]),
    writef("</listParameterType>\n").
wmpt(ParameterType) :-
    writef("<parameterType>\n"),
    writef("%d\n",[ParameterType]),
    writef("</parameterType>\n").
```

A.2.2 Signatures

A.2.2.1 Singleton

```
signature(singleton,parameters([Arg1])) :-

    Arg1 = parameter(arg_class_name,faux,[class_name]).
```

A.2.2.2 Composite

```
signature(composite,parameters([Arg1,Arg2,Arg3,Arg4])) :-

    Arg1 = parameter(composite,vrai,[class_name]),
    Arg2 = parameter(composant,vrai,[class_name]),
    Arg3 = parameter(feuilles,faux,
    [list(parameter(feuille,[class_name]),1)]),
    Arg4 = parameter(client,vrai,[class_name]).
```

A.2.2.3 Stratégie

```
signature(strategy,parameters([Arg1,Arg2,Arg3])) :-
    Arg1 = parameter(contexte,faux,[class_name]),
    Arg2 = parameter(strategy,faux,[class_name]),
    Arg3 = parameter(concreteStrategies,faux,
    [list(parameter(concreteStrategy,[class_name]),1)]).
```

A.2.2.4 Décorateur

```
signature(decorator,parameters([Arg1,Arg2])) :-
    Arg1 = parameter(composant,faux,[class_name]),
    Arg2 = parameter(decorateurConcrets,faux,
    [list(parameter(decorateurConcret,[class_name]),1)]).
```

A.2.2.5 Fabrique Astraite

```
signature(abstractFactory,parameters([Arg1,Arg2,Arg3,Arg4,
Arg5])) :-

    Arg1 = parameter(fabriqueAbstraite,faux,[class_name]),
    Arg2 = parameter(familles,faux,[list(parameter(famille,

        [literal]),1)]),
    Arg3 = parameter(client,faux,[class_name]),
    Arg4 = parameter(produits,faux,[list(parameter(produit,
    [class_name,list(parameter(produitConcret,[class_name]),

        0)]),1)]),
    Arg5 = parameter(fabriquesConcretes,vrai,
    [list(parameter(fabriqueConcrete,[class_name]),1)]).
```

A.3 DPT_Load

DPT_Load est le programme Prolog qui permet à un système externe d'interroger le noyau.

```
diagram_file('C :/exchange/Diagram.pl').
transfo_file('C :/exchange/Question.pl').
load_diagram :-
    diagram_file(DFile),
    exists_file(DFile),
    !,
    writef("loading diagram...\n\n"),
    ensure_loaded(DFile),
    writef("\ndiagram loaded.\n").
```

```
load_diagram.
call_question :-
    transfo_file(TFile),
    exists_file(TFile),
    !,
    writef("\nResolving question...\n\n"),
    ensure_loaded(TFile),
    writef("\nQuestion solved.\n").
call_question.
:-
    writef("loading sources...\n\n"),
    ensure_loaded('C :/noyau/DPT.pl'),
    ensure_loaded('C :/noyau/CD2ISL.pl'),
    ensure_loaded('C :/noyau/Tmm.pl'),
    writef("\nsources loaded.\n\n"),
    load_diagram,
    call_question,
    halt.
```

A.4 CD2ISL

CD2ISL est le programme qui permet d'exporter le diagramme de classes du *noyau* vers DB-Main, c'est-à-dire qui transcrit les clauses Prolog représentant le diagramme en données utilisables par DB-Main sous forme d'un fichier ISL.

Nous ne donnons pas le code de ce programme qui n'est pas central dans notre propos bien qu'il soit assez conséquent.

Bibliographie

[V2] *Voyager II Reference Manual Version 7 release 0* (préface de Vincent Englebert), The University of Namur - LIBD, REVER s.a., Namur, juin 2004.

[CMSEI] Carnagie Melon Software Engineering Institute, `http://www.sei.cmu.edu/`.

[TLFi] Jacques Dendien (conception et réalisation informatique), *Le Trésor de la Langue Française informatisé*, `http://atilf.atilf.fr/tlf.htm`.

[GoFDP] Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides, *Design Patterns - Catalogue de modèles de conception réutilisables*, Vuibert, Paris, 1999.

[BD] Jean-Luc Hainaut, *Ingénierie des Bases de données, Troisième édition*, Facultés Universitaires Notre Dame de la Paix, Namur, septembre 2001.

[PdP2] Jean-Marie Jacquet, *Théorie des Langages, paradigmes de programmation (2e partie)*, Programmation Logique, Facultés Universitaires Notre Dame de la Paix, Namur, 2003.

[PdP1] Baudouin Le Charlier, *Principes des Langages de Programmation : Paradigmes de Programmation (I)*, Programmation Orientée Objet, Facultés Universitaires Notre Dame de la Paix, Namur, 2000-2001.

[IUML] Pierre-Alain Muller, *Instant UML*, Wrox Press Ltd., Birmingham, 2001.

[LPP] Ulf Nilsson, Jan Maluszynski, *LOGIC, PROGRAMMING AND PROLOG (2ED)*, `http://www.ida.liu.se/~ulfni/lpp`, Ulf Nilsson and Jan Maluszynski, Linköping, 2000.

[OMG] Object Management Group, `http://www.omg.org`.

[MDAG] Object Management Group, OMG. *MDA Guide Version 1.0.1*, document number : mg/2003-06-01 edition, 2003.

[UML] Object Management Group, OMG. *OMG Unified Modeling Langage Specification version 1.5*, document number : formal/03-03-01, mars 2003.

[ATL] The Atlas Group, The Atlas Transformation Language Home Page, `http://www.sciences.univ-nantes.fr/lina/atl`, Université de Nantes, Nantes, consulté le 28 mai 2006.

[SWI] Jan Wielemaker, *SWI-Prolog 5.5 Reference Manual Updated for version 5.5.21*, `http://www.swi-prolog.org`, University of Amsterdam, Amsterdam, juillet 2005.